INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

PROGRAMMABLE ULTRASOUND COLOR FLOW SYSTEM

Ravi Managuli

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

University of Washington

2000

Program Authorized to Offer Degree: Electrical Engineering Department

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature R. A. Mainguy

Date 8/17/00

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Ravi Managuli

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

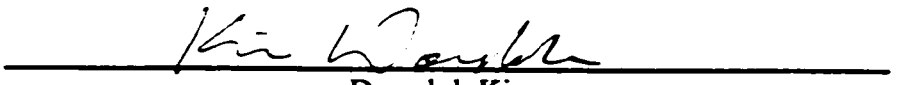
Chair of Supervisory Committee:

Yongmin Kim

Reading Committee:

Yongmin Kim

John Sahr



Donglok Kim

Date: August 9, 2000

University of Washington

Abstract

PROGRAMMABLE ULTRASOUND COLOR FLOW SYSTEM

Ravi Managuli

Chairperson of the Supervisory Committee: Professor Yongmin Kim
Departments of Bioengineering and Electrical Engineering

Ultrasound color flow systems are widely used in medical imaging because these are safe, noninvasive, and relatively inexpensive and displays images in real time. However to meet the real time requirement, these systems have been built with fixed function hardware, i.e., specialized electronic boards. This hardwired approach hinder the development of innovative algorithms to enhance the image quality and developing new applications to improve the diagnostic capability since incorporating a new application/algorithm is quite expensive, requiring redesigns ranging from hardware chips up to complete boards or some times even the complete system. On the other hand, a programmable system could be reprogrammed to quickly adapt to new tasks and offer advantages, such as reducing costs and the time-to-market of new ideas. Despite these benefits, a completely programmable color flow system that meets the real-time requirement has not been possible due to the limited computing power, inadequate data flow bandwidth or topology, algorithms not optimized for the architecture of programmable processors. This research has addressed these issues by developing a multiprocessor architecture capable of handling the computation and data flow requirements for a real-time system utilizing new generation VLIW processors, and by designing efficient ultrasound algorithms tightly integrated with the underlying architecture.

These new generation VLIW processors can deliver increased computing performance through on-chip and data-level parallelism. Even with such a flexible and

mapping of algorithms that can make good use of the available parallelism. We developed several algorithm-mapping techniques for the efficient implementation of ultrasound algorithms utilizing both on-chip and data-level parallelism. We then designed a low-cost, high-performance multiprocessor architecture capable of meeting the real-time requirements of the ultrasound color flow system. To demonstrate this multiprocessor architecture and algorithms meet the real-time requirements, we developed a multiprocessor simulation environment with a board-level VHDL simulator. Our simulation results indicate that the two-board system with 4 MAP1000s on each board is capable of supporting all the ultrasound color flow system requirements. Thus, we have demonstrated that a fully programmable ultrasound system can be developed with the reasonable number of programmable processors.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Yongmin Kim, for providing me this research opportunity. This dissertation and the research described herein would never have been completed if it were not for his encouragement, patience, energy, and his requirements for excellence. His dedication to perfection and professionalism will continue to be a lasting example and inspiration for me. I am also thankful to my supervisory committee, Donglok Kim, John Sahr, Jenq-Neng Hwang, and David Stern, who provided their insight and experience throughout this project. I am extremely grateful to my research partner, George York, whose research experience, thought-provoking conversations, kindness, generosity, and patience made this a memorable and pleasurable experience. I would like to thank several friends, who helped me in keeping my sanity while working the endlessly long hours on this research. I would also like to thank the rest of the Image Computing Systems Laboratory (ICSL) for their support and fellowship. Credit is also due to Siemens Medical Systems Ultrasound Group for funding our research and providing their technical expertise.

Dedication

I dedicate this dissertation to my family and my friends. They have supported me with their love and understanding.

Chapter 1: Introduction

The use of ultrasound for medical diagnoses dates back to more than 40 years i.e., since 1950, and now it has become a widespread medical imaging modality for routine use in hospital. Some of the many advantages of ultrasound systems include real-time image formation, noninvasive, non-ionizing radiation and provides rapid diagnoses. Ultrasound systems are painless, safe for the patient, and are portable, i.e., ultrasound system can be moved to patients rather than moving patients to the system. One of the exciting developments within the ultrasound system in the recent decade is the color flow imaging that displays an estimate of the blood velocity in the body in real time. The estimated blood velocity in pseudo color is overlaid upon a two-dimensional gray-scale image, thus presenting simultaneous echo and velocity information as shown in Figure 1-1. These systems have been widely used, e.g., in diagnosing heart valve problems, stenosis of veins and arteries and other hemodynamic problems. In addition, the spectrum of the blood velocity at a single location over time can be tracked (known as gated Doppler spectral estimation) and plotted in a spectrogram as shown in the bottom of Figure 1-1.

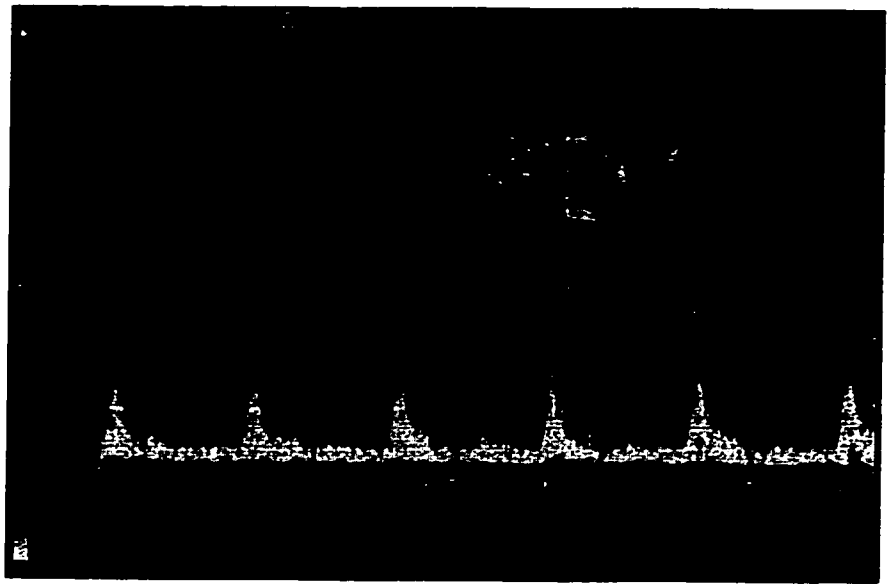


Figure 1-1. Example color flow image of the carotid artery and the corresponding spectral Doppler spectrogram.

A large amount of computing power is needed for displaying the image in real time in the ultrasound machines. The high data rates as well as the enormous computing power required have restricted the designs to algorithm-specific hardware with limited programmability. Although these specialized hardware boards may meet the speed and algorithm specifications, it lacks the flexibility and adaptability. This hardware design approach also faces a severe problem when ultrasound machines need to be upgraded with new algorithms. To accommodate each improvement/advancement in imaging and processing technology, old boards typically have to be discarded and a new machine with different boards may have to be developed. Thus, a system built with the software-centered programmable boards is much more desirable than one built with the multiple hardwired boards.

This research addresses the need of a programmable ultrasound color flow system by designing a software-centered architecture based upon the programmable digital signal processors known as mediaprocessors. These mediaprocessors employ instruction-level and data-level parallelism to achieve high computational power. We carefully mapped the

various ultrasound algorithms to the mediaprocessor architecture, creating new efficient algorithm implementations in the process. The performance of the mapped algorithms provided us with the understanding of the number of processors and the data flow required in implementing an entire system, leading to the final design of our multi-mediaprocessor architecture. Finally, to demonstrate that the system could meet the system requirements, we developed a multiprocessor simulation environment with reduced simulation complexity and time, while preserving the accuracy. Our simulation results show that a cost-effective, programmable architecture utilizing eight mediaprocessors is feasible for ultrasound processing.

The remainder of the chapter reviews the basic ultrasound processing requirements, previous programmable ultrasound systems, motivating the need for a new programmable system, and summarizes the contributions of this research.

1.1 Ultrasound system

Figure 1-2 shows a typical diagnostic ultrasound color flow system. The ultrasound acoustic signals are generated by converting pulses of 2 to 10 MHz electrical signal (known as the carrier frequency, ω_c) from the transmitter into a mechanical vibration using a piezoelectric transducer. As the acoustic wave pulse travels through the tissue, a portion of the pulse is reflected whenever material of different acoustical impedance is encountered, creating a returned signal that highlights features, such as tissue boundaries along a fairly well-defined beam line. The reflected pulses are sensed by the transducer, converted into radio frequency (RF) electrical signals and passed onto the beamformer, which are then sampled at a conservatively high rate using A/D converters. The RF demodulator then removes the carrier frequency using quadrature demodulation to recover the echo signal. The quadrature demodulation results in complex samples containing both the magnitude and phase information of the signal needed to detect moving objects, such as blood flow. The samples of the signal obtained from one acoustic pulse are called a vector. The transducer emits the acoustic pulses at a pulse repetition frequency (*PRF*) typically ranging from 0.5 to 20 kHz, based on the time for the pulse to

travel to the maximum depth and return to the transducer. Depending on how these vectors are processed, the image can either be a gray-scale image of the tissue boundaries (known as echo imaging or B-mode), a pseudo-color image, where the color represents the speed and direction of blood flow (known as color mode) or a spectral Doppler image in which a spectrum of frequencies at one location is displayed over time. In Figure 1-2 the echo processor is responsible for generating the B-mode image, the color flow processor is responsible for color flow image and the spectral Doppler processor is responsible for Doppler image. These modes are discussed in detail in chapter 2 along with the algorithms involved in generating these images.

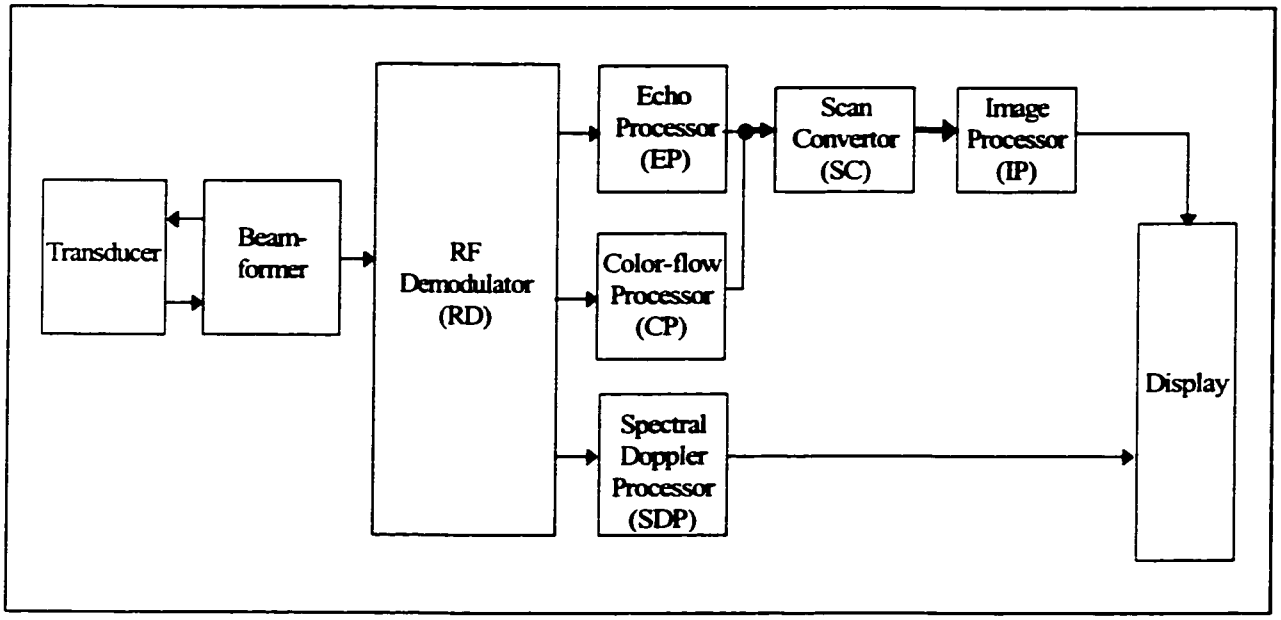


Figure 1-2. Ultrasound color flow system.

In addition to these existing modes, extensive research is still going on to improve the quality of diagnosis further. Researchers and clinicians in medical ultrasound have been exploring exciting concepts and new applications, e.g., image enhancement, three-dimensional (3D) color flow image acquisition and display, panoramic imaging, harmonic imaging, pulse inversion imaging, quantitative color flow, and improve the accuracy of blood velocity estimation. Thus, in the future ultrasound machine, we

hypothesize that there will be new modes, e.g., 3D, and new image processing options for better understanding and better analysis of the acquired image.

1.2 Computational requirements

Large amount of computing power is required to support all the processing in color flow imaging. In our recent article [1], we estimated the total computation required ranges from 31 to 55 billion operations per second (BOPS), depending on whether transcendental functions are implemented in lookup tables or calculated on the fly. The ultrasound systems are typically implemented in a hardwired fashion by using application specific integrated circuits (ASIC) and custom boards to meet the real-time requirements. To incorporate new features, such as advanced image processing applications, panoramic imaging or 3D imaging, will require even more computing power in future machines. These new applications are currently not well defined and are continually evolving. These dynamic applications will require the flexibility to adapt to changing requirements offered by programmable processors, which the hardwired ASIC approach cannot support easily.

1.3 Previous research in programmable systems

While a programmable approach offers more flexibility than the hardwired approach, an embedded programmable system capable of meeting all the computational requirements of an ultrasound machine currently has not been emerged. Many researchers developed their own "add-on" programmable systems (typically external to existing ultrasound machines) for the flexibility when developing new algorithms or applications. These systems often have difficulty achieving real-time performance, as they are not integrated with the machine and process the data off-line. Examples include off-line external systems for speckle reduction [2], intravascular ultrasound image subtraction [3], 3D reconstruction [4], and contour detection [5]. Systems designed for color flow imaging experiments require much more computing power. Jensen et al. [6] developed a programmable system with 16 Analog Devices ADSP21060 processors to estimate the

velocity of blood using autocorrelation technique. Bohs et al. [7] developed a unique system that combined several hardwired boards to perform compute-intensive velocity estimation using the sum of absolute difference method and one programmable board that utilized 2 Texas Instruments TMS320C30 DSPs and one TMS34020 graphics processor to perform postprocessing on the velocity estimated data. In addition to these experimental systems, the programmable approach is beginning to appear in commercial ultrasound machines. Siemens in collaboration with the University of Washington integrated a programmable ultrasound image processor (PUIP) board, composed of two TMS320C80 DSP processors, along with the other hardwired boards inside their Elegra ultrasound machine [8]. The ATL HDI-1000 is a mid-range ultrasound machine where programmable processors replace 50% of the previous hardware components [9]. There are also some low-end PC-based ultrasound machines emerging, supporting only B and M modes, e.g., Medison SA-5500 that uses Pentium processors combined with hardwired ASICs [10].

The PUIP board has clearly demonstrated the advantage of the programmable approach's flexibility. A new application (not initially intended for the PUIP) called panoramic imaging was quickly developed [11]. Panoramic imaging allows the user to see organs larger than the field of view of the standard B-mode sector by blending multiple images into a larger panoramic image as the multiple images are acquired. The PUIP was capable of handling panoramic imaging's real-time processing requirements of registration, warping, and interpolation. Since the exact algorithms for panoramic imaging were initially undefined, the ability to modify the programs and iterate the design was critical to quickly prototype, test and finalize the application. A hardwired design approach could not have adapted this quickly, and there would have been difficulties in creating a working prototype in a reasonable time and cost. This programmable system also has successfully proven the advantage of hardware reuse. The same hardware has been reprogrammed to offer other features in addition to panoramic imaging, such as color panoramic imaging to view blood flow in an expanded fashion, automatic fetal head measurement [12], fetal abdomen and femur measurement, harmonic

imaging, and 3D imaging [13]. However, the PUIP board was not designed to implement the entire ultrasound processing. Rather, the Elegra relies on hardwired boards for many functions, such as echo processing, color flow processing, and scan conversion.

We hypothesize that fully programmable ultrasound systems will be the future trend, allowing new innovative algorithms throughout the machine, from image enhancement [2], and new velocity estimation techniques [14] to new display modes like 3D [13] and quantitative color flow imaging [15], to be easily developed, clinically tested, which would expand the diagnostic capability of medical ultrasound machines.

1.4 Research goals and contributions

Since the color flow system includes echo processing, color flow processing, scan conversion, spectral Doppler, and raster/image processing, none of the programmable systems developed so far could meet all the computational requirements. These systems typically supported only one of many requirements and imposed limitations such as image size, ensemble size, result approximations, etc. This dissertation addresses the issues associated with proving the feasibility of a programmable ultrasound color flow system that can support all the algorithms and system requirements. The overall goal is to design a programmable color flow system that can seamlessly integrate within the ultrasound machine, provide high computational throughput and be adaptable to facilitate clinical innovations. Features include:

- ***Native***: It should be integrated inside the ultrasound machine, receiving data directly from the RF demodulator and outputting to the host computer for image display.

- ***Multi-modal***: It can process B-mode, M-mode, color-mode, power-mode, and spectral Doppler.

- ***Real time***: The processing power should be enough to be able to keep up with the data rate from the transducer (20 kHz PRF) for a dual-beam system.

- ***Cost effective***: Design should be based on a standard board, composed of commercial processors, a standard bus structure (PCI) and using standard memory components (SDRAM).

- ***Scalable***: The system can scale from a low-end to high-end system depending on the number of boards, and should have room for future expansion to support other applications, such as 3D, panoramic, quantitative, and harmonic imaging.

- ***CINE Memory***: A video sequence consisting of many image frames should be stored in the local processor memory before the filtering stages. This allows the clinician to modify the amount of speckle reduction, edge enhancement, persistence, color thresholds during CINE playback to view the sequence from a different perspective.

To meet the above mentioned system requirements and system features, we need an image processing engine with very high computational capabilities. New class of advanced DSPs, known as mediaprocessors, have recently evolved to handle the high computation requirements of multimedia applications. These provide high performance through both instruction-level and data-level parallelism. For high performance, efficient mapping of the ultrasound algorithms to the underlying mediaprocessor architecture and the multi-mediaprocessor architecture is essential. Otherwise, we may not be able to achieve the cost-effective system. In the process, we have established a methodology for mapping algorithms to mediaprocessors and developed several new algorithm implementations. Finally, to demonstrate that the system requirements are met, we developed a unique multiprocessor simulation environment using VHDL to simulate various processing modes on our ultrasound processing system.

1.5 Overview of thesis

Chapter 2 specifies the requirements that the programmable color flow system must support and describes the algorithms utilized for generating the color flow image.

Chapter 3 describes the architecture of several VLIW processors with examples. Since it is not possible to obtain high performance by directly implementing algorithms on these processors, we delineate several methodologies for obtaining the good

performance. The reason for using one particular mediaprocessor called MAP1000 for our architecture is also presented.

Chapter 4 describes the unique algorithm implementations for several ultrasound algorithms on the MAP1000 mediaprocessor for high performance. Without these unique implementations, the system could not have been able to meet the performance and cost-effectiveness requirements.

Chapter 5 summarizes the performance of algorithms on a single MAP1000. It presents the multiprocessor architecture that is necessary to meet the real time requirement of a color flow system. It discusses the simulation tools and techniques we developed and the results of our multiprocessor VHDL simulations.

Finally in Chapter 6, we summarize our conclusions and contributions of this dissertation and discuss future directions.

Chapter 2: Ultrasound Color Flow System Requirements

In designing the programmable color flow system, we paid careful attention for specifying the right system requirements and to develop the right system architecture. To arrive at the right system requirements, we reviewed literature on ultrasound imaging and their processing algorithms, evaluated various ultrasound modes, and had extensive interaction with the industry. In the next few sections, we present the system requirements, system features and color flow system image processing algorithm requirements.

2.1 ***System requirements***

Correctly specifying the system requirements is critical, as it drives the design process and determines the characteristics of the final architecture. Under specifying would result in a system incapable of handling the frame rates or quality expected by the user, while over specifying would result in an expensive system, having too many processors utilized poorly.

The system should not only meet the worst case processing requirement but also meet the data flow requirement. In addition, it should be able to accommodate new innovations and new applications either on the existing system by software development or by upgrading the system. Based on a review of the literature and the anticipated features and applications, we developed system requirements listed in Table 2-1. These requirements are driven by the worst case PRF of 20 kHz and dual beam and assume that the B-mode and color flow images have the same depth, thus the same PRF. Scenarios 1 through 4 have the worst case data flow requirement while scenarios 5 through 7 have the worst case computational requirement and scenario 8 is for Doppler processing. The largest number of samples per vector is assumed to be 1024 for B image and 512 for color flow image, with 16 bits per sample. In C mode, the system is designed to

simultaneously meet the requirement of B and color flow images at different frame rates. A slow color flow frame rate can miss details of higher frequency events, such as in the cardiac cycle. Since the color flow frame rate can be much less than the B frame rate (K > 1), this results in different temporal and spatial relationships between the two images in the output image sequence and could lead to a confusing diagnosis, as the B image could be in diastole while the color flow image is in systole. Thus most of the systems have K = 1. By providing the support for K = 4, we can implement new applications without adding any new hardware.

Table 2-1. Worst case scenarios for various processing modes.

Scenario	Mode	k	Color fps	B fps	# Color vectors	# B vectors	E	ROI	Output Image	sector angle
1	B	---	---	68.0	---	512	---	---	800x600	136
2	C	1	9.0	9.0	256	340	16	100%	800x600	90
3		2	8.4	16.8						
4		4	7.3	29.3						
5	C	1	22.3	22.3	256	256	6	100%	800x600	90
6		2	19.5	39.1						
7		4	15.6	62.5						
8	Doppler	1					256			78

2.2 Color flow ultrasound system algorithms

The various processing stages utilized in forming the ultrasound image after the reflected signal is collected by the transducer are shown in Figure 2-1. The beamformer is responsible for collecting the data from the multiple channels of the transducer and passing them onto the next stage of processing called RF demodulator. In the next few sections, we discuss signal/image processing algorithms involved in each processing stage in detail.

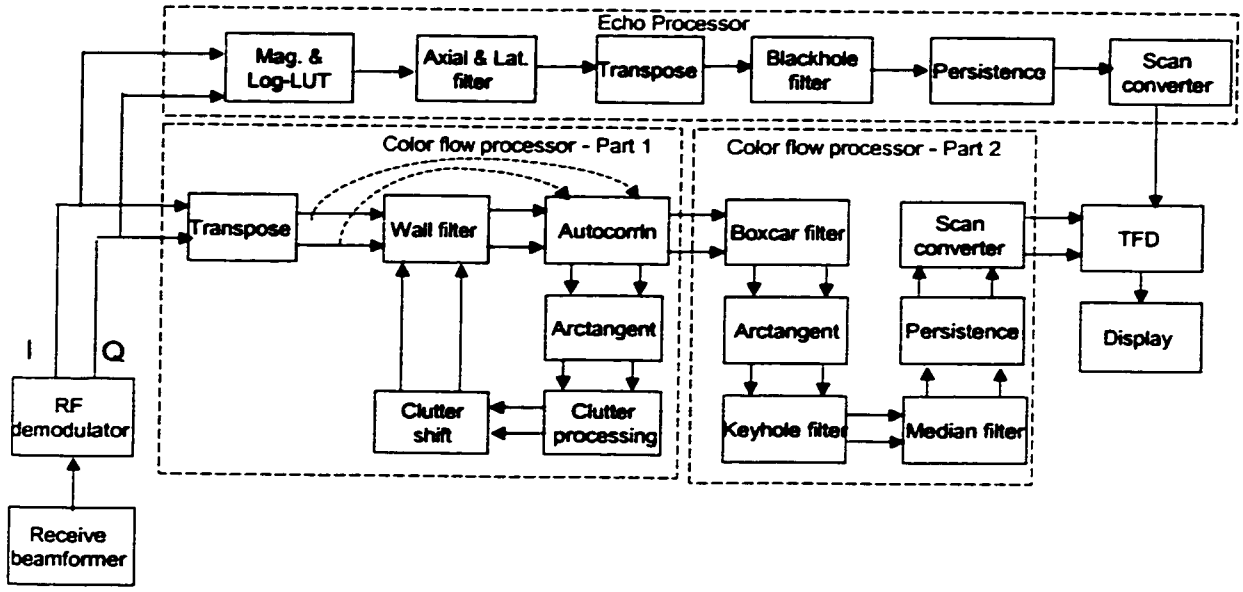


Figure 2-1. The block diagram and key algorithms in a color flow system.

2.2.1 RF demodulator

The reflected acoustic signal is a combination of the carrier frequency used for transmission, signals due to stationary tissue reflection, and Doppler signals from moving tissue and blood. Since the component of the received acoustic signal due to the carrier frequency contains no information about the body structures being investigated, it is removed by performing demodulation and then lowpass filtering the signal. To perform the demodulation, the input signal is multiplied by a cosine wave at the carrier frequency. However to determine the direction of the blood flow, i.e., to determine whether blood is flowing towards or away from the transducer, quadrature signals are necessary. Thus, to obtain the quadrature component, the reflected signal is demodulated with the sine signal as well since demodulation with cosine signal only yields the in-phase component. Both of these signals are now a combination of the desired demodulated signal at baseband and the non-desired carrier signal modulated to twice the carrier frequency. To obtain only the desired signal at baseband, the modulated signal is filtered out with a lowpass filter.

$$I_f(t) + jQ_f(t) = \sum_{n=0}^{N-1} (c(n) \cdot (I(t-n) + jQ(t-n))) \qquad (2\text{-}1)$$

where *c(0)* .. *c(N-1)* are the filter coefficients, I and Q are the in-phase and quadrature signals. Once the signal is at baseband, not as many samples are needed to preserve the signal. Thus, *I(t)* and *Q(t)* are decimated to represent the baseband signal with fewer number of samples. Since the frequencies of the signal get attenuated differentially as it travels into the body [14], there will be some artifacts in the image formation. Thus, to remove the artifacts, depth-dependent FIR filters are used after decimation. An FIR filter similar to Eq. (2-1) is utilized with an exception that the filter coefficients are selected/changed based upon the depth. The I and Q signals are then sent to the color flow and echo processor. Most of the RF demodulator algorithms are streamlined and can be executed efficiently using existing hardwired ASICS. Due to this fact, not incorporating this module into our programmable system would not lower the advantages of the programmable architecture, especially during the transition period of next 5 to 10 years. Instead, we will have a large computational advantage since this module requires a large amount of computation.

2.2.2 Echo processor

The echo processor is responsible for generating the B-mode image. The B-mode image is created by first taking the magnitude (envelope detection) of the quadrature signal, $B_a(t) = \sqrt{I^2(t) + Q^2(t)}$. Then, the signal is logarithmically compressed, $B_b(t) = \log(B_a(t))$, to reduce the dynamic range from the sampled range (around 12 bits) to that of the output display (8 bits) and to nonlinearly map the dynamic range to enhance the darker gray levels at the expense of the brighter gray levels [16]. Edge-enhancing filters are used to sharpen the tissue boundaries. A finite impulse response (FIR) can be used:

$$B_{out}(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} h(n,m) \cdot B_{in}(x-n, y-m) \qquad (2\text{-}2)$$

with the highpass filter coefficients, *h(n,m)*. These filters also enhance the noise in the image, which is typically dominated by speckle. There are several speckle filtering techniques, of which nonlinear filters, such as median filters [17] are known to preserve edges while reducing the noise.

To improve the signal to noise ratio further, multiple frames are temporally filtered [18]. Temporal filtering assumes that the frame rate is low enough to ensure the speckle is uncorrelated. It averages the current image I_{in} with the previous temporally filtered output image I_{out}, i.e., $I_{out}(k) = a \cdot I_{out}(k-1) + (1-a) I_{in}(k)$ where k is the frame number, and a is the weight (or persistence). Temporal filtering, however, would cause streaking in fast moving objects [19]. To avoid streaking, temporal filtering is made inversely proportional to the difference between the new and previous samples for the same vector, i.e., $a = f\left(1/\left|I_{in}(k) - I_{out}(k-1)\right|\right)$.

*2.3 **Spectral Doppler processor***

While B-mode imaging is useful for observing the spatial relationship between tissue layers in the body and monitoring moving structures, such as the heart and fetus, it cannot be used in visualizing faster motion, such as blood flow. In pulsed ultrasound systems, the velocity of a sample volume is estimated from:

$$v = \frac{c \cdot PRF}{4\pi f_c \cos\theta} \Delta\phi \tag{2-3}$$

where c is the velocity of sound in blood, θ is the Doppler angle (angle subtended between the transducer and the blood vessel), v is the velocity of blood, $\Delta\phi$ is the phase difference between two consecutive pulses, and f_c is the transducer center frequency. Since frequency is the derivative of phase with respect to time ($f = PRF \cdot \Delta\phi$), velocity is proportional to the frequency shift. When observing multiple signals received from a single sample volume containing multiple moving scatterers, a spectrum of frequencies can be detected, corresponding to the flow in the sample volume. Therefore, gated

Doppler spectral estimation tracks the Doppler frequencies at a single spatial location over time by collecting a large number of ensembles (e.g., N = 256 samples) at one location, then it performs a fast Fourier transform (FFT) [20].

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j(2\pi/N)nk} \quad where \quad x = I + jQ \tag{2-4}$$

Plotting the spectrum of velocities vertically versus time horizontally creates a spectrogram, as shown in Figure 1-1.

2.4 Color flow processor

While spectral Doppler can accurately quantify the blood flow at a specific location, it cannot be efficiently used to visualize the velocity profile over a region, as color flow imaging can. It is the responsibility of the color flow processor to generate the color flow image. In color flow mode, the velocity of the blood flow for a specified region of interest (ROI) is mapped to a pseudo-color image and overlaid on top of a 2D B-mode image generated by the echo processor as shown in Figure 1-1. The magnitude and direction of the velocity toward and away from the transducer (i.e., axial direction) are displayed as the brightness of colors, typically red and blue, respectively. The current velocity estimation techniques used in diagnostic ultrasound can be divided into two main categories: phase-shift and time-shift techniques. In the phase-shift technique, the velocity is estimated based on the phase shift from signals at a fixed depth while in the time-shift technique, a moving window is employed to track blood movement over a changing depth to estimate the velocity. Both techniques are based on the same physical principle in that the gradual translation of the backscattered signal with respect to the previous pulse return is due to the changing distance between the transducer and a group of moving scatterers. Most commercially-available systems use the phase-shift technique [21]. This is due to the large amount of computation required by the time-shift estimation techniques, which must process at the RF data rate while the phase-shift techniques process at the baseband data rate. We estimate the demodulation and velocity estimation

subsystems require about 32 BOPS (billion operations per second) for the time-shift technique using cross correlation and about 7 BOPS for the phase-shift technique using autocorrelation. The time-shift technique has performed well in estimating the velocity in the laboratory set up while working with the phantoms and more extensive research still need to be performed in the clinical environment objectively evaluating the improvement over the phase-shift technique.

The phase-shift technique estimates velocity mostly via autocorrelation [22]. The basis of the autocorrelation technique is that the relative phase between sequential echo vectors from stationary targets does not change with time, whereas sequential echo vectors from moving targets have corresponding changes in their relative phase. Figure 2-2 shows the relative phase change that occurs between sequential vectors due to the moving target. Multiple vectors are acquired over time (i.e., V_e, V_{e+1}, and V_{e+2} in Figure 2-2) along a single scan line with the transducer stationary, to estimate the average change in phase at each range bin. Barber et al. [23] showed that computing the first lag of autocorrelation is sufficient to correctly estimate the change in phase, $\Delta\phi$. This change in phase can be related back to velocity in Eq. (2-3). $\Delta\phi$ can be computed from:

$$\Delta\phi = \arctan\left(\frac{\mathrm{N}}{\mathrm{D}}\right) \tag{2-5}$$

where D and N are the real and imaginary part of the first lag of autocorrelation:

$$\mathrm{D} = \frac{\sum_{e=0}^{E-2}(I(e)I(e+1)+Q(e)Q(e+1))}{E-1} \tag{2-6}$$

$$\mathrm{N} = \frac{\sum_{e=0}^{E-2}(Q(e)I(e+1)-I(e)Q(e+1))}{E-1} \tag{2-7}$$

E is the ensemble size or the number of vectors shot along the same scan line, normally varying from 4 to 16. The phase-shift technique utilized to estimate the velocity suffers from two limitations, i.e., limited axial resolution and the maximum detectable velocity measurement.

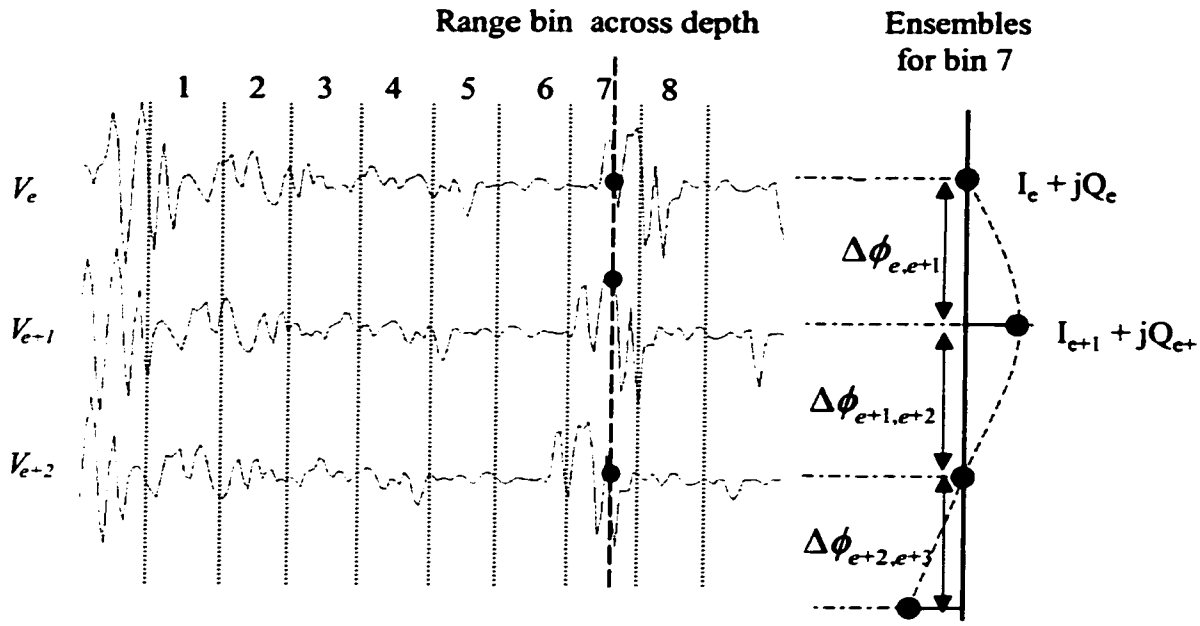


Figure 2-2. Phase shift difference between consecutive vectors due to moving particle

The transmitted signal does not have a single frequency, but a spectrum of frequencies. The attenuation of the ultrasound signal as it travels through the tissue is frequency-dependent, with larger attenuation for the higher frequency components. As a result, the center frequency of the received signal is shifted down compared to the originally-transmitted f_c [14]. This frequency shift is more dramatic for broadband signals, resulting in an increase in the variance of the velocity estimate since f_c is in the denominator of Eq. (2-3). Thus, narrowband signals are used to reduce the variance of the velocity estimate when using the phase-shift technique. This is at the expense of lower axial resolution, as broadband signals have better axial resolution (AR) given by:

$$AR = \frac{N}{4}\frac{c}{f_c} \tag{2-8}$$

where N is the number of cycles in the pulse [24]. A narrowband signal (i.e., $N = 8$ to 15) is normally used for the phase-shift technique [14], thus limiting the axial resolution (for

B-mode, $N = 2$). Also, the phase-shift technique is limited to the maximum axial velocity that can be unambiguously measured:

$$v_{max} = \frac{c}{4}\frac{PRF}{f_0} \tag{2-9}$$

The time-shift technique can overcome the axial resolution limitation by using broadband signal and the maximum velocity limitation by performing cross correlation across several range bins [25]. However, the phase-shift technique is still popular as discussed before to estimate the velocity since the computational advantages associated with the phase-shift technique far outweigh the disadvantages.

In addition to the desired signal from the blood scatterers, the received signal contains clutter noise returned from the surrounding tissue and slowly-moving vessel walls. The frequency components due to wall motion are low, e.g., < 1 kHz, while the blood motion frequency is typically around 15 kHz [26]. Due to the smooth structures of the walls, the clutter signal can be much stronger (i.e., about 40 dB) than the scattered signal from the blood [27]. In the presence of clutter, blood velocity estimates will be incorrect. Thus, many highpass filters have been developed to remove the unwanted clutter signal and improve the velocity estimates. These techniques include stationary echo cancelling, finite impulse response (FIR), infinite impulse response (IIR), and regression filters [14]. Of these techniques, regression filters have shown to have better performance compared to other techniques [28].

Since the clutter is normally not centered around zero frequency either due to the vessel wall movement in the body or the transducer movement, a filter designed with the assumption that clutter is centered on zero will not be able to filter clutter effectively. Thus, the color flow processor incorporates the adaptive wall filtering. The frequency of the clutter is estimated first, and then this frequency is utilized to shift the whole signal so that clutter gets centered on zero frequency using a complex multiplication:

$$(I(k) + jQ(k))e^{-i\theta_c k} \tag{2-10}$$

where the ensemble $k = 0,1,2,...E-1$ and θ_c is the estimated frequency of the clutter [30]. Thus, wall filtering is an adaptive two-pass feedback filter. In the first pass, the wall filter

is bypassed. The autocorrelation generates an approximation of clutter stored as estimates of the signal power, signal in-phase (denominator) and signal quadrature (numerator). The lowpass filter (clutter processor) performs axial and lateral averaging of the estimated clutter components. The revised clutter estimate is then input to the wall filter to adjust the shape for effective removal of the clutter signal. It is during the second pass that the wall filter is performed. Thus, the flow of processing in the color flow processor is *autocorrelation → clutter processing → velocity estimation → clutter shift→ wall filter → autocorrelation → CF Part 2*. Sometimes when the velocity of blood is low and close to the velocity of clutter, adaptive filtering is not performed since it might shift the color flow signal as well. In this case, the flow of processing is *wall filter → autocorrelation → CF Part 2*.

In *CF Part 2*, blood velocity is estimated and several filters are utilized to smooth out the velocity signal. A 3 x 3 2D boxcar filter (where all the filter coefficients are 1) is utilized to smooth the numerator and denominator after which velocity is estimated. The purpose of the keyhole filter is to zero out low magnitude velocity estimates. The input to the keyhole filter is the estimated velocity phasor in the polar form. The keyhole filter compares the incoming phasors to the preset threshold phasors. The phasors, which are below the threshold phasors, are zeroed. This prevents slow magnitude/velocity component to affect the color flow map for the display. The disadvantage of the keyhole filter is that it leaves a lot of holes in the image. Thus, a 3 x 3 median filter is utilized to fill the holes. The median phase is then output in the 3 x 3 neighborhood. Sudden movement of the probe or the sudden movement of the patient would cause flash of color in the image. Such flashes are distracting and they do not pertain to biologically pertinent flow. The persistence algorithm slowly decays the sudden changes when the underlying signal suddenly drops out. To maintain a persistence in case of a sudden dropout or increase in the flow signal, a decayed value for each color phasor from the previous vector is calculated and compared with the current value of the phasor. If the current phasor has a smaller phase or power than the decayed phasor calculated from the

previous phasor, then the decayed phasor is output instead of the current one (thus the flow persists by the decay constant in case of sudden dropout).

2.5 Scan conversion and tissue flow decision

After the B-mode and color flow data are acquired and processed, the polar coordinate data must then be spatially transformed to the geometry and scale of the sector scan on the Cartesian raster output image through a process known as scan conversion [31]. As Figure 2-3 shows, when a sector scan is made using a curvilinear transducer, each Cartesian raster pixel value $P(x,y)$ must be interpolated from its surrounding polar vector data $V_\psi(r)$. A 4 x 2 interpolation is normally utilized for scan conversion output.

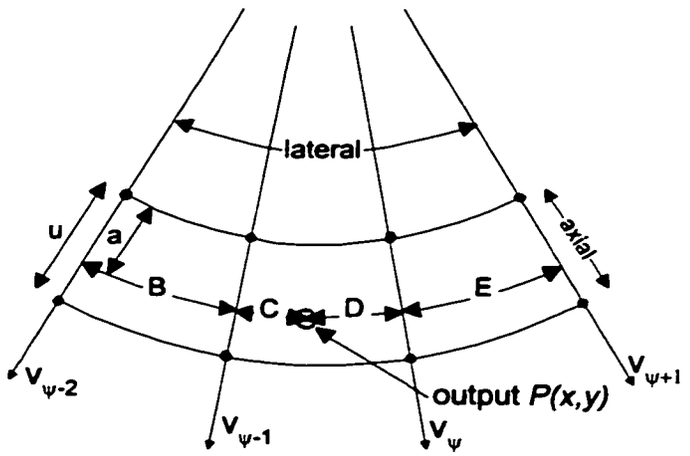


Figure 2-3. Computing an output pixel value via a 4x2 interpolation with the polar input data.

Following the creation of the B-mode and color flow frames after scan conversion, the final image processing stage contains the frame interpolation (FI) and tissue/flow decision (TFD) algorithms. If the B-mode and color flow image frames are obtained at different frame rates, then frame interpolation is used to increase the apparent frame rate of the slower mode. Bilinear interpolation is used between C_{old} and C_{new} to temporally interpolate synthetic frames. After frame interpolation, the color flow and B-mode data are at the same frame rate and can be combined into one output image. The tissue/flow decision algorithm determines whether a gray-scale B-mode pixel, $B(x,y)$, or a

color flow pixel, $\phi(x,y)$, should be output for each pixel in the color flow region of interest (ROI). The decision is based on predetermined thresholds for various parameters, and the algorithm implemented is:

If $B(x,y) > B_{threshold}$ *then*
    $Out(x,y) = B(x,y)$
Else if $|\phi(x,y)| > \phi_{threshold}$ *and* $R(x,y) > R_{threshold}$ *then*
    $Out(x,y) = \phi(x,y)$
Else
    $Out(x,y) = B(x,y)$

This is typically the last stage of processing in current ultrasound machines. In future machines, this stage could also perform more advanced image processing, such as automatically segmenting images and creating different views of several obtained images, e.g., panoramic imaging and 3D imaging.

2.6 ***Conclusion***

Large amount of computation power is required to support all these algorithms and system requirements because of which ASICs have been used as discussed before. Our goal is to design a programmable system using mediaprocessors that can meet the entire algorithm and system requirement.

New generation DSP processors are emerging to support real-time applications like digital TV, set-top boxes, desktop video conferencing, multi-function printers, digital cameras, machine vision, and medical imaging at a low cost. These mediaprocessors achieve high performance using both instruction-level and data-level parallelism. Instruction-level parallelism allows multiple operations to be initiated in a single clock cycle while data-level parallelism allows the same operation to be performed on multiple data sets simultaneously. Several factors make the VLIW architecture especially suitable for our ultrasound applications. First, most ultrasound algorithms are dominated by data-parallel computation and consist of core tight loops (e.g., convolution) that are executed repeatedly. Since the program flow is deterministic, it is possible to develop and map a new algorithm efficiently by utilizing the on-chip parallelism to its maximum. Second, single-chip high-performance VLIW processors with multiple functional units (e.g., add,

multiply and load/store) have become commercially available recently at a low cost. In the next chapter, we present VLIW processor architecture and their features in detail with a few example processors.

Chapter 3: VLIW Architecture for DSP Applications

In this chapter, both architectural and programming features of VLIW processors are discussed. In the next section, VLIW's architectural features are outlined while several commercially-available VLIW processors are discussed in Section 3.2. Algorithm mapping methodologies on VLIW processors are presented in Sections 3.3 and why we choose the MAP1000 for our architecture is presented in Section 3.4.

3.1 VLIW architecture

A VLIW processor has a parallel internal architecture and is characterized by having multiple independent functional units [34] as shown in Figure 3-1. Factors that make VLIW processors powerful are: (1) instruction-level parallelism (2) data-level parallelism (3) powerful instruction set and (4) programmable data flow. Each of these factors is discussed in the following sections.

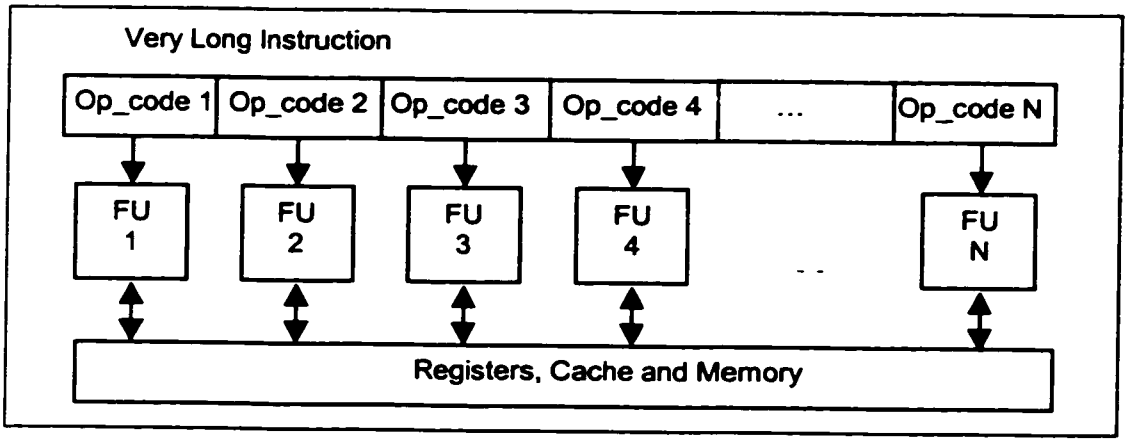


Figure 3-1. Block diagram for a typical VLIW processor with multiple functional units.

3.1.1 Instruction-level parallelism (ILP)

The programs can be sped up by executing several RISC-like operations, such as load, stores, multiplications and additions, all in parallel on different functional units. Each

very long instruction contains an operation code for each functional unit, and all the functional units receive their operation codes at the same time. Thus, VLIW processors typically follow the same control flow across all functional units. The register file and on-chip memory banks are shared by multiple functional units. A better illustration of ILP is provided with an example. Consider the computation of

$$y = a_1x_1 + a_2x_2 + a_3x_3$$

on a sequential RISC processor

cycle 1: ***load*** a_1
cycle 2: ***load*** x_1
cycle 3: ***load*** a_2
cycle 4: ***load*** x_2
cycle 5: ***multiply*** $z_1\ a_1\ x_1$
cycle 6: ***multiply*** $z_2\ a_2\ x_2$
cycle 7: ***add*** $y\ z_1\ z_2$
cycle 8: ***load*** a_3
cycle 9: ***load*** x_3
cycle 10: ***multiply*** $z_1\ a_3\ x_3$
cycle 11: ***add*** $y\ y\ z_2$

which requires 11 cycles. On the VLIW processor that has two load/store units, one multiply unit and one add unit, the same code can be executed in only 5 cycles.

cycle 1: ***load*** a_1
load x_1
cycle 2: ***load*** a_2
load x_2
multiply $z_1\ a_1\ x_1$
cycle 3: ***load*** a_3
load x_3
multiply $z_2\ a_2\ x_2$
cycle 4: ***multiply*** $z_3\ a_3\ x_3$
add $y\ z_1\ z_2$
cycle 5: ***add*** $y\ y\ z_3$

Thus, the performance is approximately two times faster than that of a sequential RISC processors. If this loop needs to be computed repeatedly (e.g., FIR), the free slots

available in cycles 3, 4, and 5 can be utilized by overlapping the computation and loading for the next output value to improve the performance further.

3.1.2 Data-level parallelism

Also, the programs can be sped up by performing partitioned operations where a single arithmetic unit is divided to perform the same operation on multiple smaller precision data, e.g., a 64-bit ALU is partitioned into eight 8-bit units to perform eight operations in parallel. Figure 3-2 shows an example of *partitioned-add* where eight pairs of 8-bit pixels are added in parallel by a single instruction. This feature is often called as multimedia extension [35]. By dividing the ALU to perform the same operation on multiple data, it is possible to improve the performance by 2x, 4x or 8x depending upon the partition size. The performance improvement using data-level parallelism is also best explained with an example of adding two arrays (*a* and *b* each has 128 elements with each array element being 8 bits), which is shown below.

```
/* Each element of array is 8 bits */
char a[128], b[128], c[128];
for (i = 0; i < 128; i++){
    c[i] = a[i] + b[i];
}
```

The same code can be executed utilizing *partitioned_add* :

```
long a[16], b[16], c[16];
for (i = 0; i < 16; i++){
  c[i] = partitioned_add(a[i], b[i]);
}
```

The performance with data-level parallelism is increased by a factor of eight in this example. Since the number of loop iterations also decreases by a factor of eight, there will be an additional performance improvement due to the reduction of branch overhead.

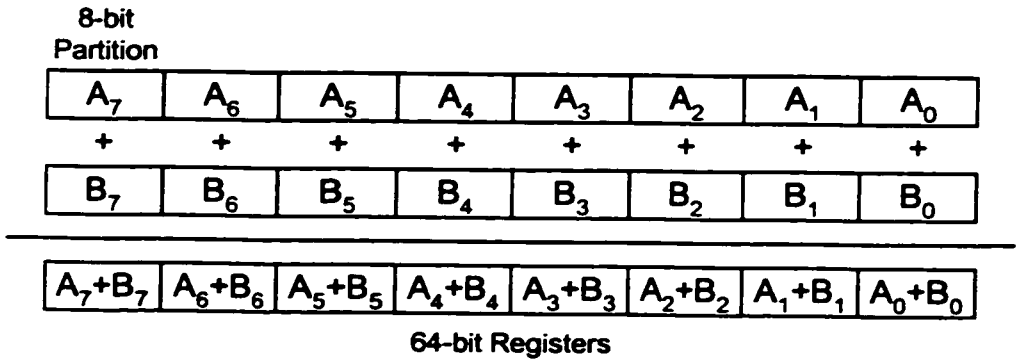


Figure 3-2. Example partition operation: *partitioned-add.*

To support data-level parallelism well, the functional units (e.g., ALU and multiplier) have to be designed accordingly with a well-designed instruction set.

3.1.3 Instruction set architecture

The data-level parallelism in multimedia applications can be utilized by a special subset of instructions, called Single Instruction Multiple Data (SIMD) instructions [36-37]. These instructions operate on multiple 8, 16, or 32-bit sub-words of the operands. The current SIMD instructions can be categorized into the following groups:

- Partitioned arithmetic/logic instructions: *add*, *subtract*, *multiply*, *compare*, *shift* and etc.
- Sigma instructions: *inner-product*, sum of absolute difference (*SAD*), sum of absolute value (*SAM*), and etc.
- Partitioned select instructions: *min/max*, *conditional_selection*, and etc.
- Formatting instructions: *map*, *shuffle*, *compress*, *expand*, and etc.
- Processor-specific instructions optimized for multimedia, imaging and 3D graphics.

The instructions in the first category perform multiple arithmetic operations in one instruction. The example of *partitioned_add* is shown in Figure 3-2, which performs the same operation on eight pairs of pixels simultaneously. These partitioned arithmetic/logic units can also saturate the result to the maximum positive or negative value, truncate the

data or round the data. The instructions in the second category are very powerful and useful in ultrasound processing. Eq. (3-1) is an *inner-product* example while Eq. (3-2) describes the operations performed by the *SAD* instruction, where x and c are eight 8-bit data stored in each 64-bit source operand and the results are accumulated in y.

$$y = \sum_{i=0}^{i=7} c_i * x_i \tag{3-1}$$

$$y = \sum_{i=0}^{i=7} | c_i - x_i | \tag{3-2}$$

The *inner-product* instruction is ideal in implementing convolution-type algorithms while the *SAD* and *SAM* instructions are very useful in video processing, e.g., motion estimation. The third category of instructions can be used in minimizing the occurrence of *if/then/else* to improve the utilization of instruction-level parallelism. The formatting instructions in the fourth category are mainly used for rearranging the data in order to expose and exploit the data-level parallelism. An example of using *shuffle* and *combine* to transpose a 4 x 4 block is shown in Figure 3-3. Two types of *compress* are presented in Figure 3-4. *compress1* in Figure 3-4(a) packs two 32-bit values into two 16-bit values and stores them into a partitioned 32-bit register while performing the right-shift operation by a specified amount. *compress2* in Figure 3-4(b) packs four 16-bit values to four 8-bit values while performing the right-shift operation by a specified amount. *compress2* saturates the individual partitioned results after compressing to 0 or 255. In the fifth category of instructions, each processor has its own instructions to further enhance the performance. For example, *complex_multiply* shown in Figure 3-5, which in one instruction performs two partitioned complex multiplications, is useful for implementing the FFT and autocorrelation algorithms.

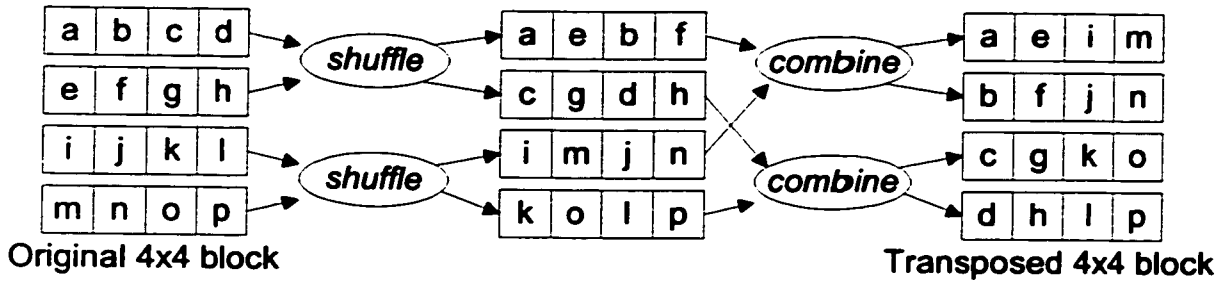


Figure 3-3. Transpose of a 4x4 block using *shuffle* and *combine*.

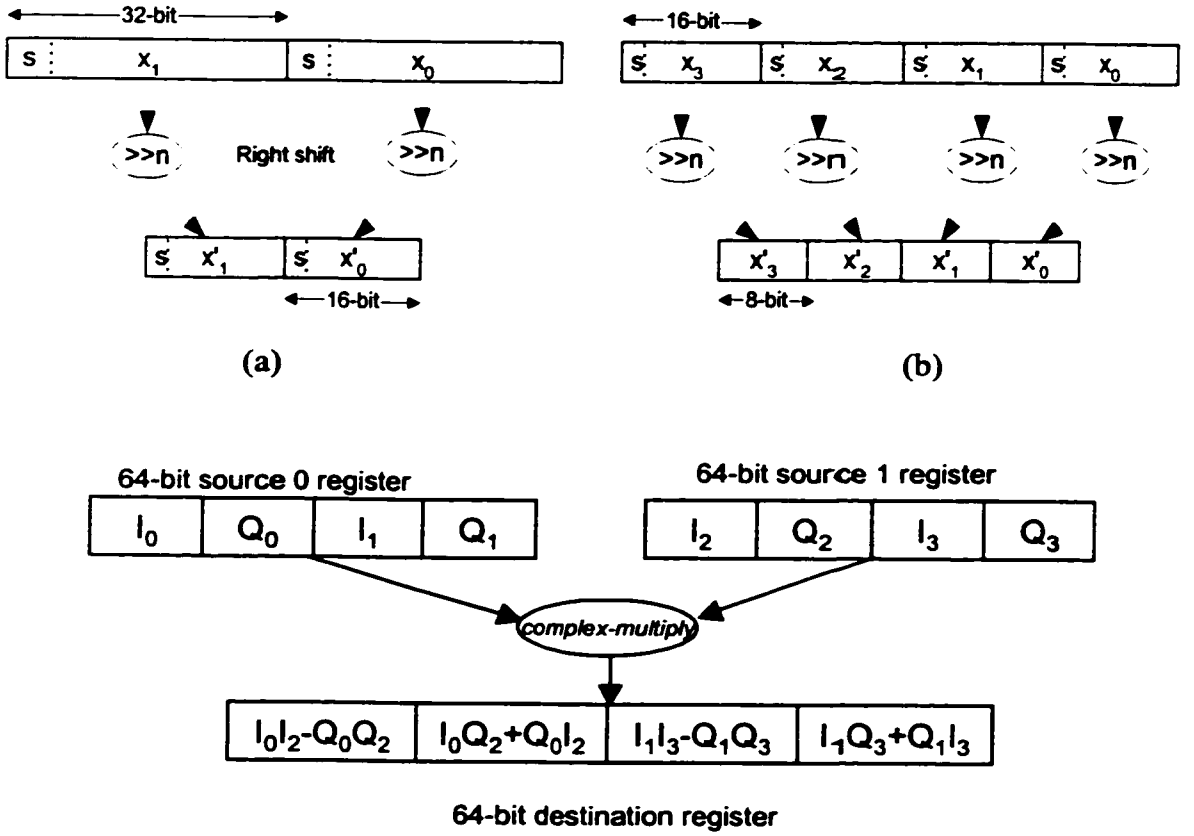


Figure 3-4. Partitioned 64-bit *compress* instructions.

Figure 3-5. *complex-multiply* instruction.

Although these instructions are powerful, they take multiple cycles to complete, which is defined as latency. For example, partitioned arithmetic/logic instructions have a

3-cycle latency while sigma instructions have a latency of 5 to 6 cycles. To achieve the best performance, all the execution units need to be kept as busy as possible every cycle, which is difficult due to these latencies. However, these instructions have a single-cycle throughput (i.e., another identical operation can be issued in the next cycle) due to hardware pipelining. In Section 3.3.2 loop unrolling and software pipelining are discussed, which tries to exploit this single-cycle throughput to overcome the latency problem.

Many improvements in the processor architectures and powerful instruction sets have been steadily reducing the processing time, which makes the task of bringing the data from off-chip to on-chip memory fast enough so as not to slow down the functional units a real challenge. This problem gets exasperated with the growing speed disparity between the processor and the off-chip memory, e.g., the number of CPU cycles required to access the main memory doubles approximately every 6.2 years [38].

3.1.4 Memory I/O

There are several methods to move the data between slower off-chip memory and faster on-chip memory. The conventional method of handling data transfers in general-purpose processors has been via data caches [36], while the DSPs have been more relying on direct memory access (DMA) controllers [39]. Data caches have a nonpredictable access time. The data access time to handle a cache miss is at least an order of magnitude slower than that of a cache hit. On the other hand, the DMA controller has a predictable access time and can be programmed to hide the data transfer time behind the processing time by making it work independently from the core processor. But it requires some effort by the programmer, e.g., the data transfer type, amount, location, and other information including synchronization between DMA and processor have to be thought through and specified by the programmer [40]. In Section 3.3.6 DMA programming techniques, to hide the data movement time behind the core processor's computing time, are presented.

Many DSP programmers have developed their applications in assembly language. However, programming in assembly language is difficult to code, debug, maintain, and

port, especially as applications become larger and more complex and processor architectures get very sophisticated. For example, the arrival of powerful VLIW processors with the complex instruction set and the need to perform loop unrolling and software pipelining has increased significantly the complexity and difficulty of assembly language programming. Thus, a lot of efforts are being made to develop intelligent compilers that can reduce or ultimately eliminate the burden and need of assembly language programming.

3.1.5 Programming of VLIW processors

For VLIW processors, the scheduling of all instructions is the responsibility of the programmer and/or compiler. Thus, the assembly language programmers must understand the underlying architecture intimately to be able to obtain high performance in a certain algorithm and/or application. Smart compilers for VLIW processors to ease the programming burden are very important. Tightly coupled with the advancement of compiler technologies, there have been many useful programming techniques to improve the performance of compiled code, such as loop unrolling, software pipelining and use of C intrinsics [41]. In particular, C intrinsics can be used as a good compromise between the performance and programming productivity. A C intrinsic is a special C language extension, which looks like a function call, but directs the compiler to use a certain assembly language instruction. In programming TMS320C62, for example, the *int _add2(int, int)* C intrinsic would generate an ADD2 assembly instruction (two 16-bit partitioned additions) using two 32-bit integer arguments. The compiler technology has advanced to the point that the compiled programs using C intrinsics in some cases have been reported to approach up to 60 - 70% of the hand-optimized assembly program performance [42].

The use of C intrinsics improves the programming productivity since the compiler can relieve the programmer from register allocation, software pipelining, handling multi-cycle latencies, and other tasks. However, what instructions to use still depends upon the programmer, and the choice of instructions decides the performance that can be obtained.

Thus, careful analysis and design of an algorithm to make good use of powerful instructions and instruction-level parallelism is essential [43]. Algorithms developed without any consideration of the underlying architecture often do not produce the desired level of performance. Another important issue that needs to be taken care of in order to obtain the good performance is the efficient data handling between the on-chip and off-chip memories. A DMA controller could be used to hide the I/O time behind the processing time as much as possible.

In the next section, several commercially available VLIW processors are briefly discussed.

*3.2 **Example VLIW processors***

Every VLIW processor tries to utilize both instruction-level and data-level parallelism. They distinguish between themselves in the number of banks and amount of on-chip memory and/or cache, the number and type of functional units, the way in which the global control flow is maintained, and the type of interconnections between the functional units. In this section, five VLIW processors and their basic architectural features are briefly discussed. Many of these processors have additional functional units to perform sequential processing, such as that required in MPEG's Huffman decoding.

3.2.1 Texas Instruments TMS320C62

The Texas Instruments TMS320C62 [44] shown in Figure 3-6 is a VLIW architecture with 256 bits per instruction. This DSP features 2 clusters, each with 4 functional units. Each cluster has its own 16 32-bit registers with two read ports and one write port for each functional unit. There is one cross-cluster read port each way, so a functional unit in one cluster can access values stored in the register file of the other cluster. Most operations have a single-cycle throughput and a single-cycle latency except for a few operations. For example, a multiply operation has a single-cycle throughput and a 2-cycle latency, while a load/store operation has a single-cycle throughput and a 5-cycle latency. Two integer arithmetic units support partitioned operations in that each 32-bit arithmetic

and logic unit (ALU) can be split to perform two 16-bit additions or two 16-bit subtractions. The TMS320C62 also features a programmable Direct Memory Access (DMA) controller combined with two 32-kbyte on-chip data memory blocks to handle I/O data transfers.

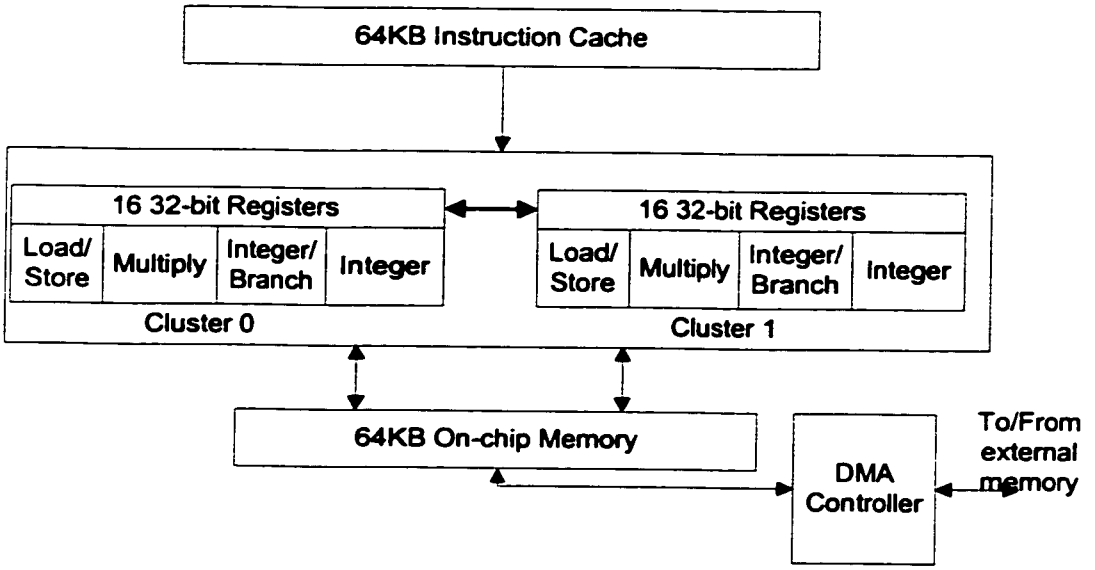


Figure 3-6. Block diagram of the Texas Instruments TMS320C62.

3.2.2 Fujitsu FR500

The bock diagram of the Fujitsu FR500 [45] VLIW processor is shown in Figure 3-7. It can issue up to four instructions per cycle. It has two integer units, two floating-point units, a 16-kbyte 4-way set-associative data cache, and a 16-kbyte 4-way set-associative instruction cache. This processor has 64 32-bit general purpose registers and 64 32-bit floating-point registers. Integer units are responsible for double word load/store, branch, integer multiply, and integer divide operations. They also support integer operations, such as rotate, shift, and AND/OR. All these integer operations have a single-cycle latency except load/store, multiply and divide. Multiply has a 2-cycle latency with a single-cycle throughput, divide has a 19-cycle latency with a 19-cycle throughput, and load/store has a

3-cycle latency with a single-cycle throughput. Floating-point units are responsible for single-precision floating-point operations, double word load and SIMD-type operations. All the floating-point operations have a 3-cycle latency with a single-cycle throughput except load, divide, and square root. Floating-point divide and square root operations have a 10-cycle and 15-cycle latency, respectively, and they cannot be pipelined with another floating-point divide or square root operation since the throughput for both of these operations is equal to their latency. For load, latency is 4-cycle while throughput is single-cycle. The floating-point unit also performs multiply and accumulate with 40-bit accumulation, partitioned arithmetic operations on 16-bit data, and various formatting operations. Partitioned arithmetic operations have either one or two-cycle latency with a single-cycle throughput. All computing units support predicated execution for *if/then/else*-type statements. Since this processor does not have a DMA controller, it has to rely on caching mechanism to move the data between on-chip and off-chip memory.

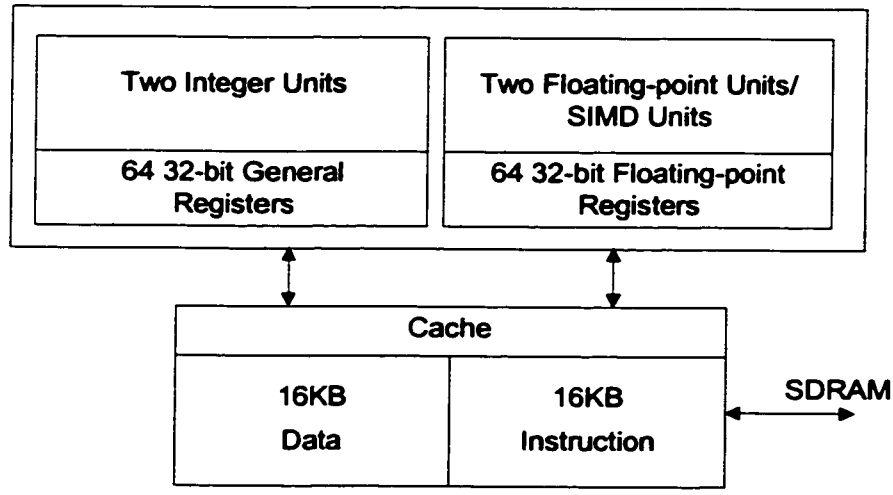


Figure 3-7. Block diagram of the Fujitsu FR500.

3.2.3 Texas Instruments TMS320C80

The Texas Instruments TMS320C80 [46] incorporates not only instruction-level and data-level parallelism, but also multiple processors on a single chip. Figure 3-8 shows the TMS320C80's block diagram. It contains four Advanced Digital Signal Processors

(ADSPs; each ADSP is a DSP with a VLIW architecture), a RISC processor, and a programmable DMA controller called Transfer Controller (TC). Each ADSP has its own 2-kbyte instruction cache and four 2-kbyte on-chip data memory modules that are serviced by the DMA controller. The RISC processor has a 4-kbyte instruction cache and a 4-kbyte data cache.

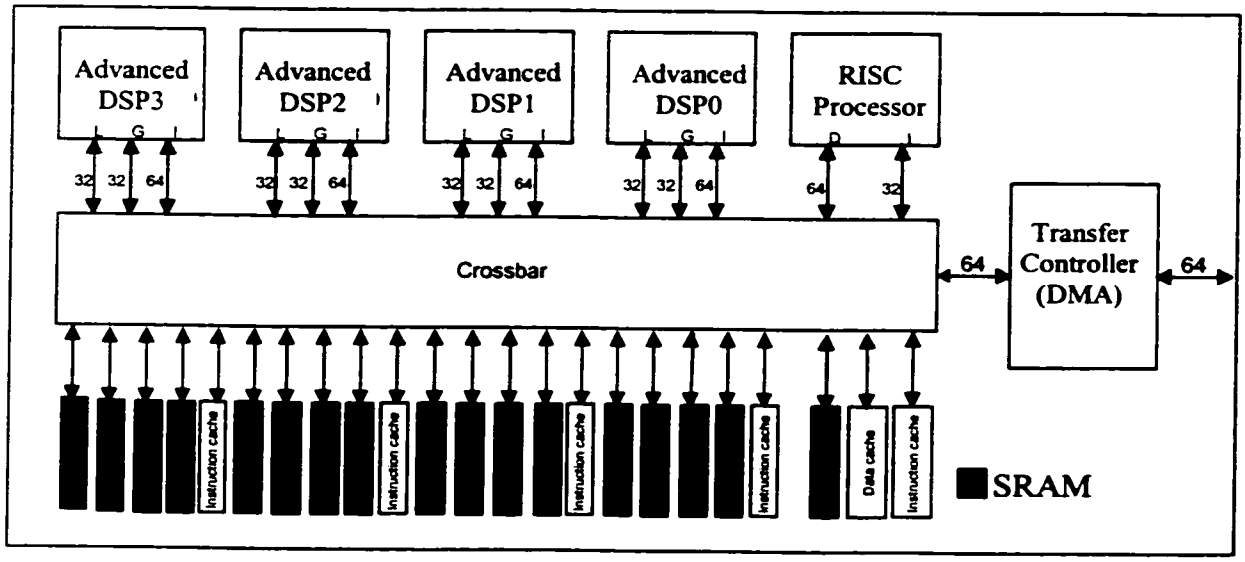


Figure 3-8. Block diagram of the Texas Instruments TMS320C80.

Each ADSP has a 16-bit multiplier, a 3-input 32-bit ALU, a branch unit, and two load/store units. The RISC processor has a floating-point unit, which can issue floating-point multiply/accumulate instructions on every cycle. The programmable DMA controller supports various types of data transfers with complex address calculations. Each of the five processors is capable of executing multiple operations per cycle. Each ADSP can execute one 16-bit multiplication (that can be partitioned into two 8-bit multiply units), one 32-bit add/subtract (that can be partitioned into two 16-bit or four 8-bit units), one branch, and two load/store operations in the same cycle. Each ADSP also has three zero-overhead loop controllers. However, this processor does not support some powerful operations, such as SAD or *inner-product*. All operations on the ADSP including load/store, multiplication and addition are performed in a single cycle.

3.2.4 Philips Trimedia TM1000

The block diagram of the Philips Trimedia TM1000 [37] is shown in Figure 3-9. It has a 16-kbyte data cache, a 32-kbyte instruction cache, 27 functional units, and various I/O units and coprocessors. The TM1000 does not have a programmable DMA controller and relies on the caching mechanism to move the data between on-chip and off-chip memory. The TM1000 can issue five simultaneous operations to five out of the 27 functional units per cycle, i.e., five operation slots per cycle. The two DSPALU units can each perform either 32-bit or 8-bit/16-bit partitioned arithmetic operations. Each of the two DSPMUL units can issue two 16×16 or four 8×8 multiplications per cycle. Furthermore, each DSPMUL can perform an inner-product operation by summing the results of its two 16×16 or four 8×8 multiplications. In ALU, pack/merge (for data formatting) and select operations are provided for 8-bit or 16-bit data in the 32-bit source data. All the partitioned operations including load/store and inner-product type operations have a 3-cycle latency and a single-cycle throughput.

FU	Quant
ALU	5
Const.	5
Memory	2
Shift	2
DSPALU	2
DSPMUL	2
Branch	3
FALU	2
IFMUL	2
FCOMP	1
FDIV/FSQRT	1

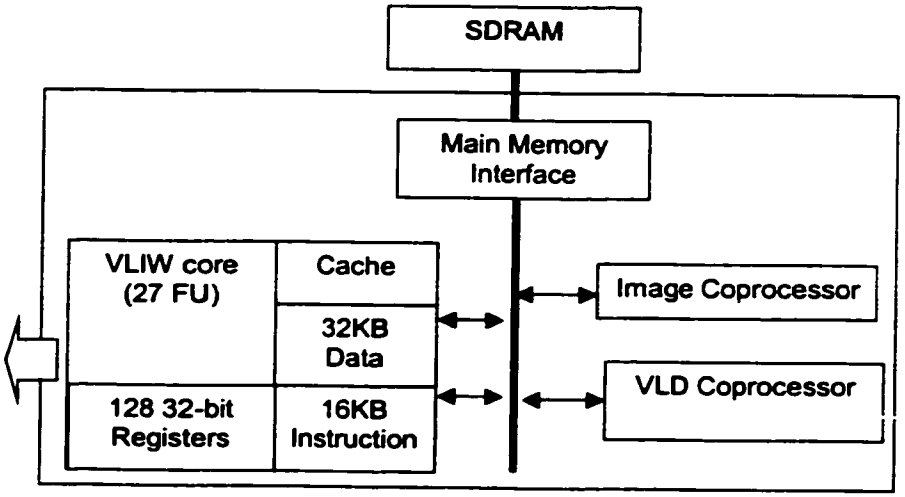


Figure 3-9. Block diagram of the Philips Trimedia TM1000.

3.2.5 Hitachi/Equator Technologies MAP1000

The block diagram of the Hitachi/Equator Technologies MAP1000 [47] is shown in Figure 3-10. The processing core consists of 2 clusters, a 16-kbyte 4-way set-associative data cache, a 16-kbyte 2-way set-associative instruction cache and a video graphics coprocessor. It has an on-chip programmable DMA controller called Data Streamer (DS). Each cluster has 64 32-bit general registers, 16 predicate registers, a pair of 128-bit registers, an Integer Arithmetic Logic Unit (IALU) and an Integer Floating-point Graphics Arithmetic Logic Unit (IFGALU). Two clusters are capable of executing four different operations (e.g., 2 on IALUs and 2 on IFGALUs) per clock cycle. The IALU can perform either a 32-bit fixed-point arithmetic operation or a 64-bit load/store operation. The IFGALU can perform 64-bit partitioned arithmetic operations, sigma operations on 128-bit registers (on partitions of 8, 16, and 32), and various formatting operations on 64-bit data (e.g., map and shuffle). The IFGALU unit can also execute floating-point operations including division and square root. Partitioned arithmetic

operations have a 3-cycle latency with a single-cycle throughput, multiply and inner-product operations also have a 6-cycle latency with a single-cycle throughput. Most of the floating-point operations also have a 6-cycle latency with a single-cycle throughput except divide and square root, which have 17-cycle latency with a 16-cycle throughput. The MAP1000 has a unique architecture in that it supports both data cache and DMA mechanism. With the DMA approach, the 16-kbyte data cache itself can be used as on-chip memory. The MAP1000's dual PCI ports will provide a good basis for interconnecting the multiple processors gluelessly.

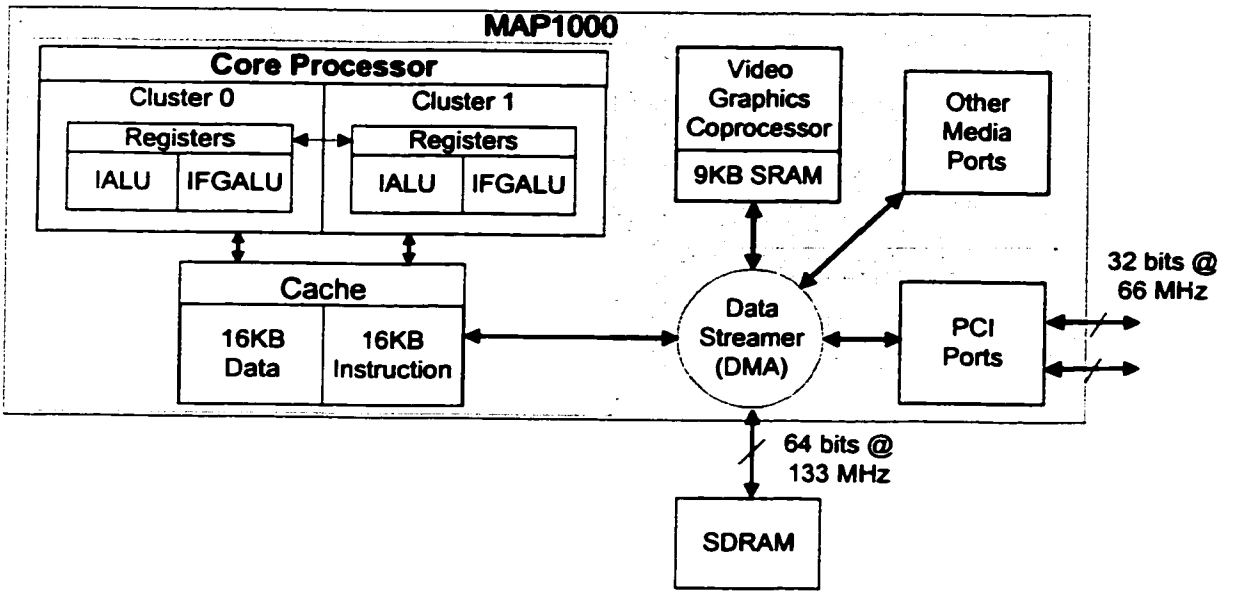


Figure 3-10. Block diagram of the Hitachi/Equator Technologies MAP1000.

3.3 Mapping of algorithms to the VLIW architecture

Implementation of an algorithm onto a VLIW processor for high performance requires the good understanding of the algorithm, processor architecture, and instruction set. There are several programming techniques that can be used to improve the algorithm performance. These techniques include:

- Judicious use of instructions to utilize multiple execution units and data-level parallelism.
- Loop unrolling and software pipelining.
- Avoiding conditional branching.
- Overcoming memory alignment problems.
- Utilizing fixed-point operations instead of floating-point operations.
- Use of the DMA controller to minimize I/O overhead.

In this section, these techniques are discussed in detail.

3.3.1 Judicious use of instructions

VLIW processors have the optimum performance when all the functional units are utilized efficiently and maximally. Thus, the careful selection of instructions to utilize the underlying architecture to keep all the execution units busy is critical. For illustration, consider an example where a lookup table operation is performed, i.e., LUT (x[i]).

```
char x[128], y[128];
for (i=0; i < 128; I++)
    y[i] = LUT(x[i]);
```

This algorithm mapped to the MAP1000 without considering the instruction set architecture requires 3 IALU operation (2 loads and 1 store) per data point, which corresponds to 384 instructions for 128 data points (assuming one cluster). By utilizing multimedia instructions so that both IALU and IFGALU are well used, the performance can be improved significantly as shown in Figure 3-11. Here, 4 data points are loaded in a single *load* IALU instruction, and the IFGALU is utilized to separate each data point before the IALU performs LUT operations. After performing the LUT operations, the IFGALU is again utilized to pack these four data points so that a single *store* instruction can store all four results. This algorithm leads to 6 IALU operations and 6 IFGALU

operations for every four data points. Since the IALU and IFGALU can run concurrently, this reduces the number of cycles per pixel to 1.5 compared to 3 before. This results in a performance improvement by a factor of 2. This simple example illustrates that it is possible to improve the performance significantly by carefully selecting the instructions in implementing the intended algorithm.

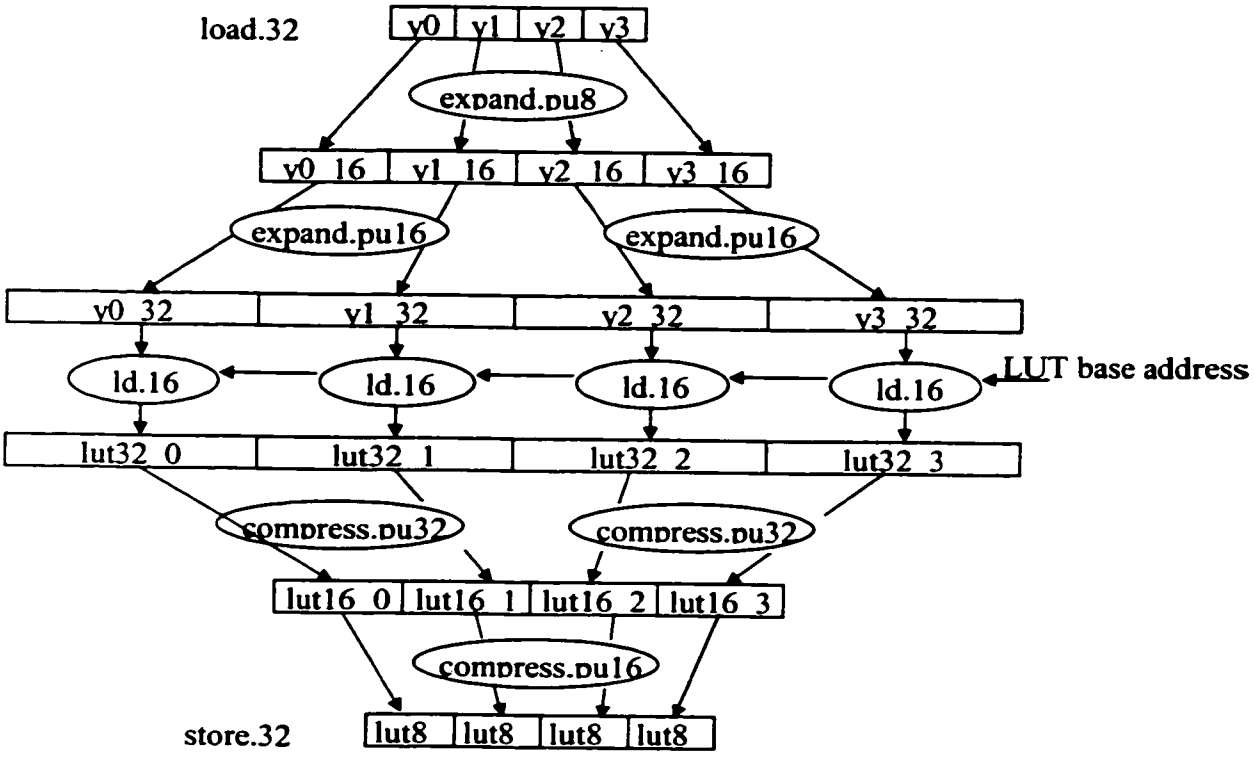


Figure 3-11. Performing LUT using IALU and IFGALU on the MAP1000.

3.3.2 Loop unrolling and software pipelining

Loop unrolling and software pipelining are very effective in overcoming the multiple-cycle latencies of the instructions. For illustration, consider an algorithm implemented on the MAP1000, where each element of an array is multiplied with a constant *k*. On the MAP100, *load/store* has a 5-cycle latency with a single-cycle throughput, and *partitioned_multiply* (that performs eight 8-bit partitioned multiplications) has a 6-cycle

latency with a single-cycle throughput. Figure 3-12(a) illustrates the multiplication of each array element (in_data) with a constant *k* where *k* is replicated in each partition of the register for *partitioned_multiply*. The array elements are loaded by the IALU, and *partitioned_multiply* is performed by the IFGALU. Since *load* has a 5-cycle latency, *partitioned_multiply* is issued after 5 cycles. The result is stored after another latency of 6 cycles since *partitioned_multiply* has a 6-cycle latency. Only 3 instruction slots are utilized out of 24 possible IALU and IFGALU slots in the inner loop, which results in wasting 87.5% of the instruction issue slots and leads to a disappointing computing performance.

To address this latency problem and underutilization of instruction slots, loop unrolling and software pipelining can be utilized [48]. In loop unrolling, multiple sets of data are processed inside the loop. For example, six sets of data are processed in Figure 3-12(b). The latency problem is partially overcome by taking advantage of the single-cycle throughput and filling the delay slots with the unrolled instructions. However, many instruction slots are still empty since the IALU and IFGALU are not used simultaneously. Software pipelining can be used to fill these empty slots, where operations from different iterations of the loop are overlapped and executed simultaneously by the IALU and IFGALU as shown in Figure 3-12 (c). By the IALU loading the data to be used in the next iteration and the IFGALU executing the *partitioned_multiply* instructions using the data loaded in the previous iteration, the IALU and IFGALU can execute concurrently, thus increasing the instruction slot utilization. Few free slots available in the IFGALU unit can be utilized for controlling the loop counters. However, to utilize software pipelining, some preprocessing and postprocessing need to be performed. For example, loading in the prologue the data to be used in the first iteration and executing *partitioned_multiply* and *store* in the epilogue for the data loaded in the last iteration as shown in Figure 3-12 (c). Thus, the judicious use of loop unrolling and software pipelining results in the increased data processing throughput when Figure 3-12 (a) and Figure 3-12 (c) are compared (a factor of 5.7 when an array of 480 is processed, i.e., 720 cycles vs. 126 cycles).

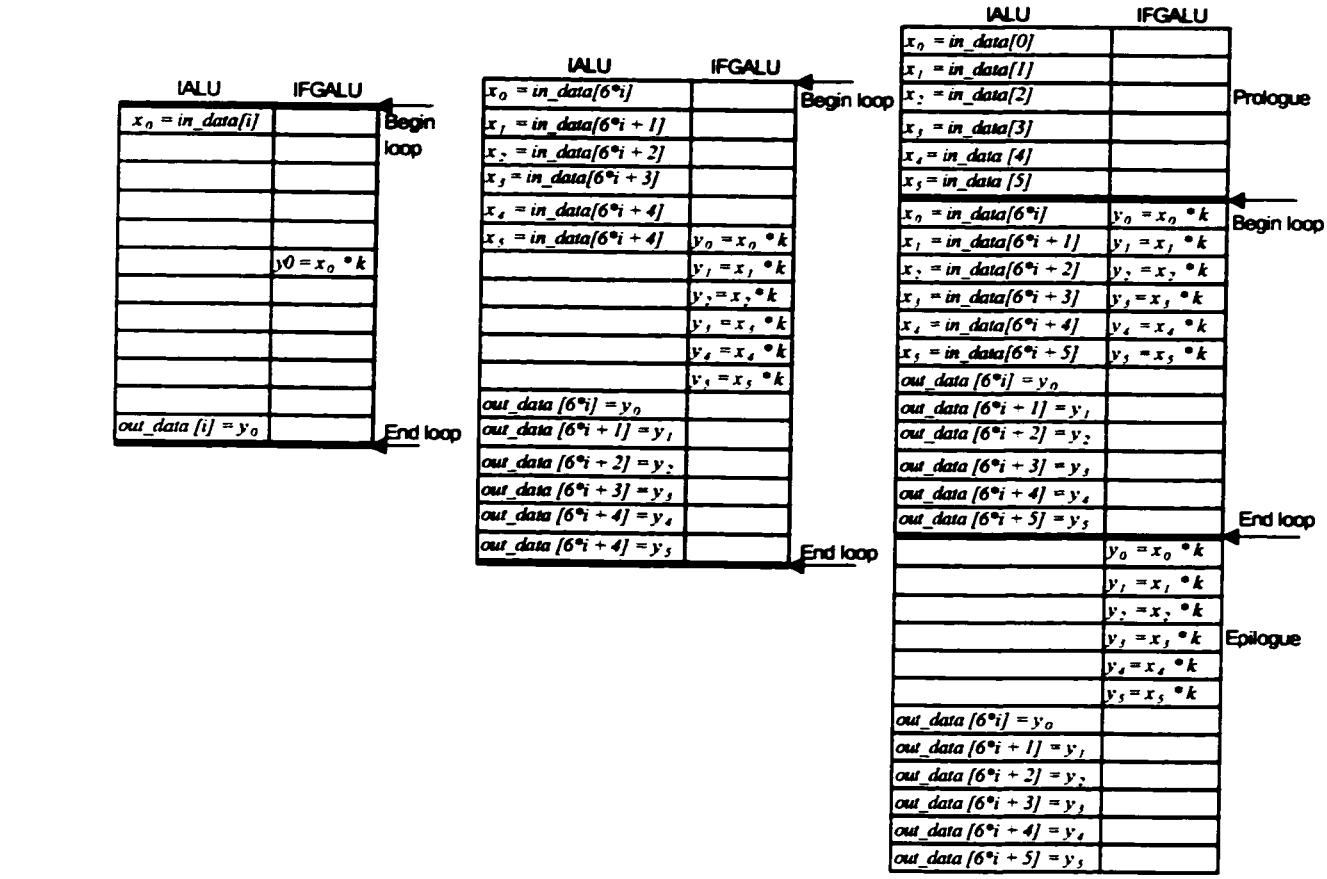


Figure 3-12. Example of loop unrolling and software pipelining.

3.3.3 Fixed-point vs. Floating-point

The VLIW processors predominantly have fixed-point functional units with some floating-point support. The floating-point operations are generally computationally-expensive with longer latency and lower throughput than fixed-point operations. Thus, it is desirable to carry out ultrasound algorithm computations in fixed-point arithmetic and avoid floating-point operations if we can.

While using fixed-point arithmetic, the programmer has to pay attention to several issues, e.g., accuracy and overflow. When multiplying two numbers, the number of bits required to represent the result without any loss in accuracy is equal to the sum of the number of bits in each operand, e.g., while multiplying two N-bit numbers 2N bits are necessary. Storing 2N bits is expensive and is usually not necessary. If only N bits are kept, it is up to the programmer to determine which N bits to keep. Several instructions on these VLIW processors provide a variety of options to the programmer in selecting, which N bits to keep.

Overflow occurs when too many numbers are added to the register accumulating the results, e.g., when a 32-bit register tries to accumulate the results of 256 multiplications each with two 16-bit operands. One measure that can be taken against overflow is to utilize more bits for the accumulator, e.g., 40 bits to accumulate the above results. Many DSPs do in fact have extra headroom bits in the accumulators (TMS320C62 and FR500). The second measure that can be used is to clip the result to the largest magnitude positive or negative number that can be represented with the fixed number of bits. This is more acceptable than permitting the overflow to occur, which otherwise would yield a large magnitude and/or sign error. Many VLIW instructions can automatically perform clip operation (MAP1000 and TM1000). The third measure is to shift the product before adding it to the accumulator. A complete solution to the overflow problem requires that the programmer be aware of the scaling of all the variables to ensure that the overflow would not happen.

If a VLIW processor supports floating-point arithmetic, it is often convenient to utilize the capability. For example, in case of computing square root, it is advantageous to utilize a floating-point unit rather than using an integer unit with a large lookup table. However, to use floating-point operations with integer operands, some extra operations are required, e.g., converting floating-point numbers to fixed-point numbers and vice versa. Furthermore, it takes more cycles to compute in floating point compared with in fixed point.

3.3.4 Avoiding if-then-else statements

There are two types of branch operations that occur while programming.

Loop branching: Most ultrasound algorithms spend a large amount of time in simple inner loops. These loops are usually iterated many times, the number of which is constant and predictable. Usually, the branch instructions that are utilized to loop back to the beginning of a loop have a minimum of a two-cycle latency and require decrement and compare instructions. Thus, if the inner loop is not deep enough, the overhead due to branch instructions can be rather high. To overcome this problem, several processors support the hardwired loop handling capability, which does not have any delay slots and does not require any decrement and compare instructions. It automatically decrements the loop counter (set outside the inner loop) and jumps out of the loop as soon as the branch condition is satisfied. For other processors that do not have a hardwired loop controller, a loop can be unrolled several times until the effect of additional instructions (decrement and compare) becomes minimal.

if/then/else branch: Conditional branching inside the inner loop can severely degrade the performance of a VLIW processor. For example, the direct implementation of the following code segment on the MAP1000 (where *X*, *Y*, and *S* are 8-bit data points) would be Figure 3-13(a), where the *branch-if-greater-than* (*BGT*) and *jump* (*JMP*) instructions have a three-cycle latency.

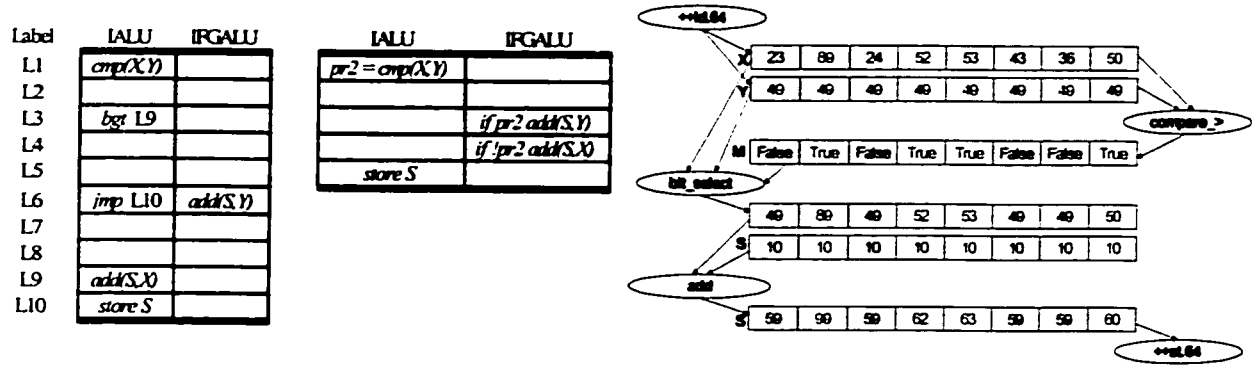
```
if (X > Y)
  S = S + X;
else
  S = S + Y;
```

Due to the idle instruction slots, it takes either 7 or 10 cycles per data point (depending on the path taken) since we cannot use instruction-level and data-level parallelism effectively. Thus, to overcome this *if/then/else* barrier, two methods can be used:

- *Use predicated instructions*: Most of the instructions can be predicated. A predicated instruction has an additional operand that determines whether or not the instruction should be executed. These conditions are stored either in a separate

set of 1-bit registers called predicate registers or regular 32-bit registers. An example with a predicate register to handle the *if/then/else* statement is shown in Figure 3-13(b). This method requires only five cycles (compared to 7 or 10) to execute the same code segment. A disadvantage of this approach is that only one data point is processed at a time, thus it cannot utilize data-level parallelism.

- *Use select instruction*: *select* along with *compare* can be utilized to handle the *if/then/else* statement efficiently. *compare* as illustrated in Figure 3-13(c) is utilized in comparing each pair of sub-words in two partitioned source registers and storing the result of the test (i.e., TRUE or FALSE) in the respective sub-word in another partitioned destination register. This partitioned register can be used as a mask register (*M*) for the *select* instruction, which depending on the content of each mask register partition, selects either *X* or *Y* sub-word. As there are no branches to interfere with software pipelining, it only requires 5 cycles per loop. Out of these 5 cycles, two IALU cycles (for load and store) can be hidden behind 3 IFGALU cycles (*compare_>*, *bit_select*, *add*) utilizing loop unrolling and software pipelining discussed in the previous section thus reducing the number of cycles per loop to 3. More importantly, since the data-level parallelism, i.e., partitioned operations, of the IFGALU is used, the performance increases further by a factor of 8 for 8-bit sub-words.



(a) Direct implementation (b) Using a predicate register (c) Using partitioned operations

Figure 3-13. Avoiding branches while implementing *if/then/else* code.

3.3.5 Memory alignment

To take advantage of the partitioned operations, the address of the data loaded from memory needs to be aligned. For example, if the partitioned register size is 64 bits (8 bytes), then the address of the data loaded from memory into the destination register should be a multiple of 8 [49]. When the input data words are not aligned, extra overhead cycles are needed in loading two adjacent data words and then extracting the desired data word by performing shift and mask operations. An instruction called *align* is typically provided to perform this extraction. Figure 3-14 shows the use of *align*, where the desired non-aligned data, x_3 through x_{10}, are extracted from the two adjacent aligned data words (x_0 through x_7 and x_8 through x_{15}) by specifying a shift amount of 3.

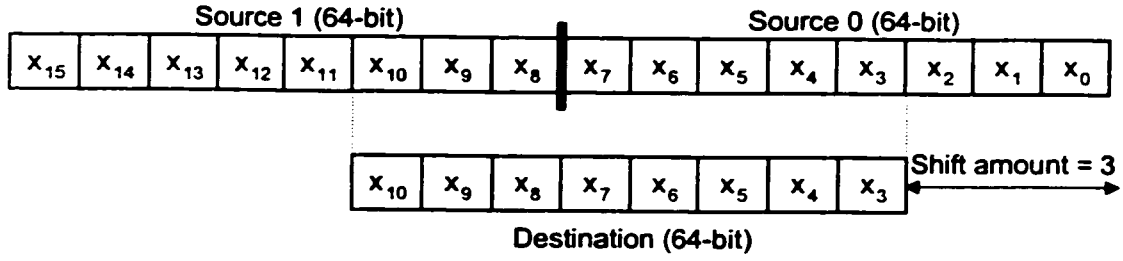


Figure 3-14. *align* instruction to extract the non-aligned eight bytes.

3.3.6 DMA programming

In order to overcome the I/O bottleneck, a DMA controller can be utilized, which can stream the data between on-chip and off-chip memory, independent of the core processor. In this section, two DMA modes frequently used are described: 2D block transfer and guided transfer. 2D block transfers are utilized for most applications while the guided transfer mechanism is utilized for some special purpose applications, e.g., lookup table or where the required data are not consecutive.

3.3.6.1 2D block transfer

In this mode, the input data are transferred and processed in small blocks as shown in Figure 3-15. To prevent the processor from waiting for the data as much as possible, the DMA controller is programmed to manage the data movements concurrently with the processor's computation. This technique is illustrated in Figure 3-15 and is commonly known as *double buffering*. Four buffers, two for input blocks (*ping_in_buffer* and *pong_in_buffer)* and two for output blocks (*ping_out_buffer* and *pong_out_buffer)*, are allocated in the on-chip memory. While the core processor computes on a current image block (e.g., block #2) from *pong_in_buffer* and stores the result in *pong_out_buffer*, the DMA controller moves the previously-calculated output block (e.g., block #1) in *ping_out_buffer* to the external memory and brings the next input block (e.g., block #3) from the external memory into *ping_in_buffer*. When the computation and data movements are both completed, the core processor and DMA controller switch buffers, with the core processor starting to use the *ping* buffers and the DMA controller working on the *pong* buffers.

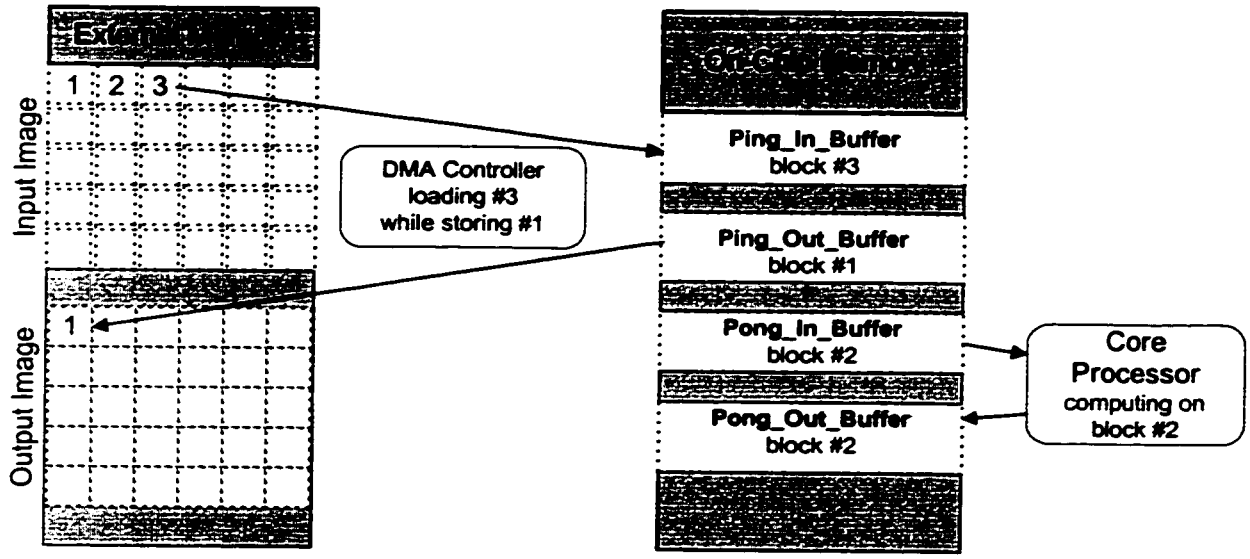


Figure 3-15. Double buffering with a programmable DMA controller.

3.3.6.2 Guided transfer

While 2D block-based transfers are useful when the memory access pattern is regular, it is inefficient for accessing the non-sequential or randomly scattered data. The guided transfer mode of the DMA controller can be used in this case to efficiently access the external memory based on a list of memory address offsets from the base address, called *guide table*. One example of this is shown in Figure 3-16. The guide table is either given before the program starts (off-line) or generated in the earlier stage of processing. The guided transfer is set up by specifying *base address*, *data size*, *count*, and *guide table pointer*. *data size* is the number of bytes that will be accessed from each guide table entry, and the guide table is pointed by *guide table pointer*.

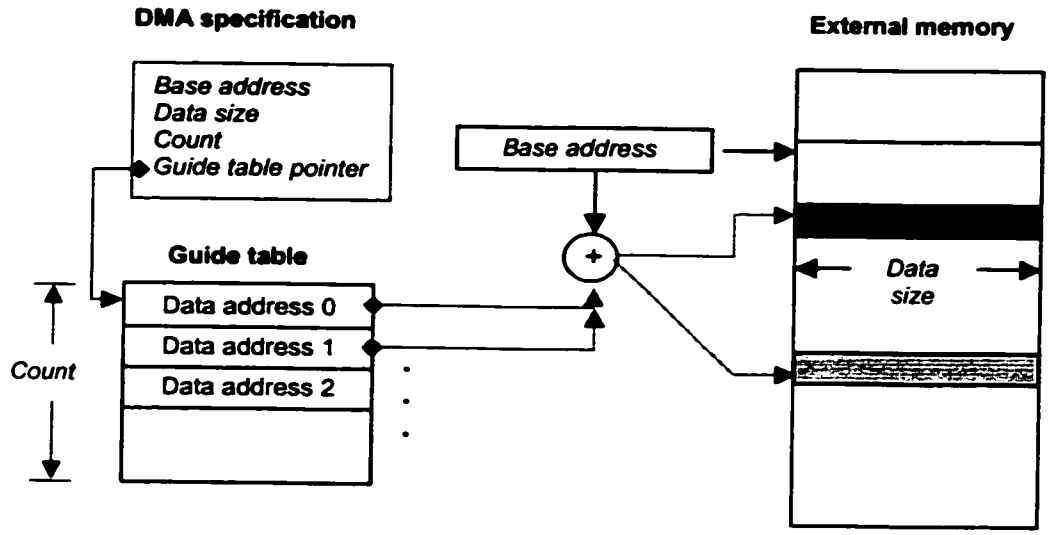


Figure 3-16. *Guided transfer* DMA controller.

3.4 Mediaprocessor selection

Features of several mediaprocessors are compared in Table 3-1. Features related to data flow capability such as the availability for on-chip DMA, the SDRAM bus bandwidth (SDRAM BW), the size of the on-chip data memory (Data RAM), and the number and bandwidth of the glueless interprocessor (IP) ports are listed in this table. Regarding computational ability, the maximum number of 8-bit additions per second (million additions per second, Madds/s), the maximum number of 16x16 *inner-product* operations (millions of multiply-accumulates per second, MMAC/s), and the maximum BOPS ratings are also shown for various processors. The number of partitioned registers available per processing cluster is listed, as a large number of registers are needed for many applications to approach their ideal performance on a processor, particularly when using techniques such as software pipelining, discussed in Section 3.3.2 The power drawn by each chip is also a consideration, as several processors will be needed to implement the system. Finally, the cost of a system will be influenced not only by the

CPU price listed, but also by the support for glueless interprocessor connectivity and the total number of processors required.

Table 3-1. Comparison of processors considered.

Processors	Fujitsu FR500	Phillips TM1100	TI TMS320C62	TI TMS320C80	Equator MAP1000
Architecture	VLIW	VLIW	VLIW	VLIW	VLIW
Clock (MHz)	266	133	200	60	200
Power (W)	2	6		10	6
Glueless IP connectivity	none	PCI 1x132 MB/s	Serial Links 2x3.1 MB/s	none	dual PCI 2x264 MB/s
DMA vs Cache	Cache	Caching	DMA	DMA	Both
SDRAM BW (MB/s)	400	532	332	480	800
Data RAM (kbytes)	16	16	64	36	16
Number of registers per cluster	128 32-bit	128 32-bit	16 32-bit	8 32-bit	32 64-bit
Processing clusters	1	1	2	4	2
Partitioned Adds (Madds/s)	1064 16-bit	1064 8-bit	800 8-bit	960 8-bit	3200 8-bit
16x16 Innerprod (MMAC/s)	1064	266	400	240	3200
Max BOPS*	4.25	2.3	2	2.8	6.4
CPU price ($)**	$50-$100	$80	$25-180	$150	$40-150

*excluding special instructions for sum-of-absolute difference

**marketing estimates, subject to change

We decided to use the MAP1000 mediaprocessor for our ultrasound system as it leads most of the categories in Table 3-1 in addition to the following reasons:

- Easier programming while maintaining high performance. For example, the C implementation for morphology on the MAP1000 was only 1.7 times slower than the ideal performance [50].
- The efficient implementation of several algorithms on the MAP1000 yielded performance that are comparable to the performance on hardwired ASICS. For example 3 x 3 2D convolution on the MAP1000 is faster than the LSI LOGIC's L64240 [50].

- The MAP1000 has many instructions optimized for multimedia, imaging and 3D graphics, which are suitable in the ultrasound applications as well.
- We have been a partner since 1994 with Hitachi Ltd. and Equator Technologies in defining the MAP1000 architecture and have extensive experience in its programming and applications.

3.5 Method to determine efficiency of algorithms

Programming in a high-level language like C is preferred for ease of programming and maintenance. To determine whether the C compiler's ability to handle software pipelining and loop unrolling is good enough for a given algorithm or assembly programming is required, we use the following methods. We first break down our computation loop into the IALU and IFGALU operations required, thus determining whether our algorithm is IALU-bound or IFGALU-bound. The maximum of the two determines the minimum number of cycles ($t_{compute}$ assuming the instruction latencies are overcome by ideal software pipelining) required to compute the number of pixels in each cluster. We also estimate the I/O time ($t_{i/o}$) required to move the input and output images on and off chip assuming the ideal bandwidth on the SDRAM bus is achieved. From these two numbers, we have an estimate as to if our algorithm is *compute-bound* (i.e., $t_{compute} > t_{i/o}$) or *I/O-bound* (i.e., $t_{i/o} > t_{compute}$). After coding and simulations, we evaluate C compiler's performance compared to our ideal time ($t_{compute}$). Our rule-of-thumb is that if an algorithm is *compute-bound* and more than 2 times slower in C compared to ideal, then we implement the loop in assembly language. If an algorithm is *I/O-bound,* it becomes a candidate for data flow sharing, as discussed in the next chapter.

In the next chapter, we discuss the implementation of several ultrasound algorithms on the MAP1000. The performance obtained using these implementations and the data flow study of several algorithms helped us to design a multiprocessor architecture suitable for ultrasound system.

Chapter 4: Ultrasound Algorithm Mapping

In this chapter, we discuss the implementation details of several key algorithms on the MAP1000. These efficient algorithms have not only provided results that can be used to more accurately estimate the number of processors required in the complete ultrasound system, but also aided in understanding the data flow and architectural requirements needed for designing the multi-mediaprocessor architecture.

4.1 2D convolution

Convolution plays a central role in echo processing, color flow processing and RF downconversion. In convolution, each output pixel is computed to be a weighted average of several neighboring input pixels. Generalized 2D convolution of an N x N input image with an M x M convolution kernel is defined as

$$b(x,y) = \frac{1}{s}\sum_{i=x}^{x+M-1}\sum_{j=y}^{y+M-1} f(i,j)h(x-i,y-j) \qquad (4\text{-}1)$$

where f is the input image, h is the input kernel, s is the scaling factor, and b is the convolved image.

The generalized convolution has one division operation for normalizing the result as shown in Eq. (4-1). To avoid this time-consuming division operation, we multiply the reciprocal of the scaling factor with each kernel coefficient beforehand and then represent each coefficient in 16-bit sQ15 fixed-point format (one sign bit followed by 15 fractional bits). With this fixed-point representation of coefficients, right-shift operations can be used instead of division. The right-shifted result is saturated to 0 or 255 for the 8-bit output, i.e., if the right-shifted result is less than 0, it is set to zero; if it is greater than 255, then it is clipped to 255; otherwise it is left unchanged.

The generic code for the 2D convolution algorithm utilizing a typical VLIW processor instruction set is shown below. It generates eight output pixels that are horizontally consecutive. In this code, the assumptions are that the number of partitions is eight (the data registers are 64 bits with eight 8-bit pixels), the kernel register size is 128 bits (eight 16-bit kernel coefficients) and the kernel width is less than or equal to eight. *align* instructions are used to extract the unaligned data from two adjacent data words. If the kernel width is greater than eight, then the kernel can be subdivided into several sections and the inner loop is iterated multiple times, while accumulating the multiplication results.

```
for (i = 0; i < kernel_height ; i++){
    /* Load 8 pixels of input data x0 through x7 and kernel coefficients c0 through c7 */
    image_data_x0_x7 = *src_ptr; kernel_data_c0_c7 = *kernel_ptr;
    /* Compute inner-product for pixel 0 */
    accumulate_0 += inner-product (image_data_x0_x7, kernel_data_c0_c7);
    /* Extract data x1 through x8 from x0 through x7 and x8 through x15 */
    image_data_x8_x15 =*( src_ptr + 1);
    image_data_x1_x8 = align(image_data_x8_x15 : image_data_x0_x7, 1);
    /* Compute the inner-product for pixel 1 */
    accumulate_1 += inner-product (image_data_x1_x8 : kernel_data_c0_c7);
    /* Extract data x2 through x9 from x0 through x7 and x8 through x15 */
    image_data_x2_x9 = align(image_data_x8_x15 : image_data_x0_x7, 2);
    /* Compute the inner-product for pixel 2 */
    accumulate_2 += inner-product (image_data_x2_x9 : kernel_data_c0_c7);
    .....
    accumulate_7 += inner-product (image_data_x7_x15 : kernel_data_c0_c7 );
    /* Update the source and kernel addresses */
    src_ptr = src_ptr + image_width;
    kernel_ptr = kernel_ptr + kernel_width;
} /* end for i */
/* Compress eight 32-bit values to eight 16-bit values with right-shift operation */
result64_ps16_0 = compress1(accumulator_0 : accumulator_1 , scale);
result64_ps16_1 = compress1(accumulator_2 : accumulator_3, scale);
result64_ps16_2 = compress1(accumulator_4 : accumulator_5 , scale);
result64_ps16_3 = compress1(accumulator_6 : accumulator_7, scale);
/* Compress eight 16-bit values to eight 8-bit values. Saturate each individual value to 0
or 255 and store them in two consecutive 32-bit registers */
result32_pu8_0 = compress2(result64_ps16_0 : result64_ps16_1, zero);\
result32_pu8_1 = compress2(result64_ps16_2 : result64_ps16_3, zero);
/* Store 8 pixels present in two consecutive 32-bit registers and update the destination
address */
*dst_ptr++ = result32_pu8_0_and_1;
```

The MAP1000 has an advanced *inner-product* instruction, as shown in Figure 4-1. It can multiply eight 16-bit kernel coefficients (of PLC) by eight 8-bit or 16-bit input pixels (of PLV) and accumulate the multiplication results into a specified 32-bit destination register. This instruction can also shift a new pixel into a 128-bit PLV register. x_0 through x_{23} represent sequential input pixels while c_0 through c_7 represent kernel coefficients. After performing an inner-product operation shown in Figure 4-1, x_{16} is shifted into the leftmost position of the 128-bit register (PLV) while x_0 is shifted out. The next time this instruction is executed, the inner-product will be performed between $x_1 \sim x_8$ and $c_0 \sim c_7$. This new pixel shifting-in capability eliminates the need of multiple align instructions used in the code above. An instruction called *setplc.128* sets the 128-bit PLC register with kernel coefficients. The MAP1000 also has *compress* instructions similar to the ones shown in Figure 3-4 that can be utilized for computing the convolution output.

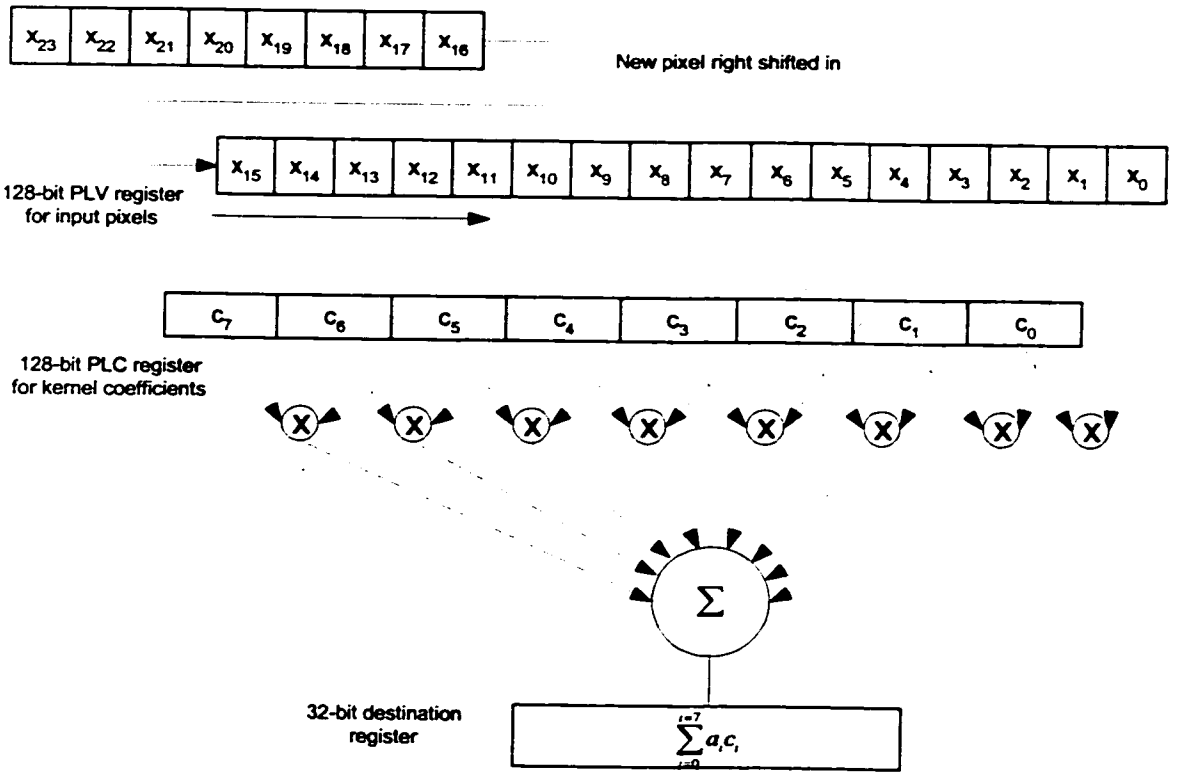


Figure 4-1. Advanced *inner-product* instruction.

The 2D convolution execution times on a 512 x 512 image with different-size kernels on the 200-MHz MAP1000 are plotted in Figure 4-2. The convolution time does not increase quadratically ($O(M^2)$) as the kernel size increases because of the availability of *inner-product*. Instead, it increases linearly since it only has to process more rows with the increasing kernel height until the kernel width reaches 8 (the number of pixels that can be processed in one *inner-product* instruction). When the kernel width exceeds 8, there is a jump in the convolution time as shown in Figure 4-2. Another jump occurs when the kernel width crosses 16. This performance is much faster than the previously-reported software-based convolution and is comparable with the hardwired implementations [52].

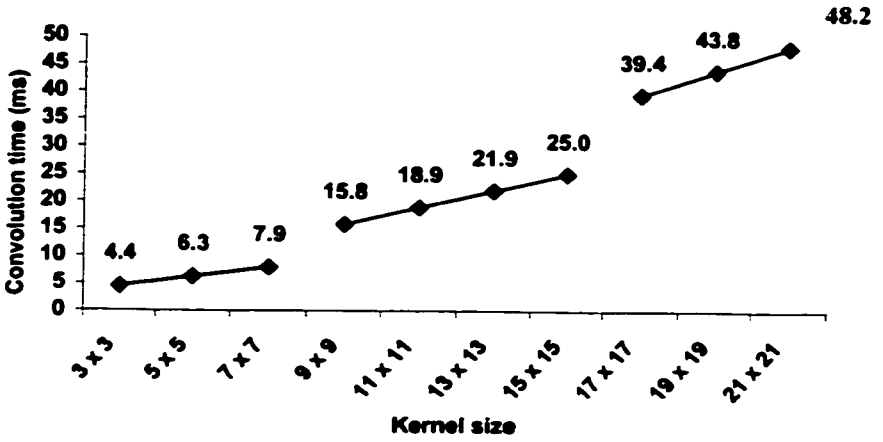


Figure 4-2. MAP1000 convolution time vs. kernel size on a 512 x 512 x 8-bit image.

There are several methods to further reduce the amount of computation in implementing convolution. Separable kernels are normally used in the ultrasound machine for filtering the data axially and laterally. Thus while using separable kernels, 2D convolution could be implemented by applying two consecutive 1D kernels (x-directional convolution on the original image followed by y-directional convolution on the x-directional convolved image), thus reducing the number of multiplications substantially [53]. In order to utilize the powerful *inner-product* instruction, the intermediate image after x-directional convolution needs to be transposed (row-column

transformation) so that another x-directional convolution (instead of y-directional convolution) can be performed. Then, one more transposition is performed before storing the final result. The performance of separable convolution along with generalized convolution on the MAP1000 is shown in Figure 4-3. Separable convolution is advantageous only for kernel sizes 9 x 9 and above. Thus for separable filters in the echo processor, it is advantageous to use 2D convolution while in RF downconverter separable convolution is useful where the kernel size can go up to 17 x 17.

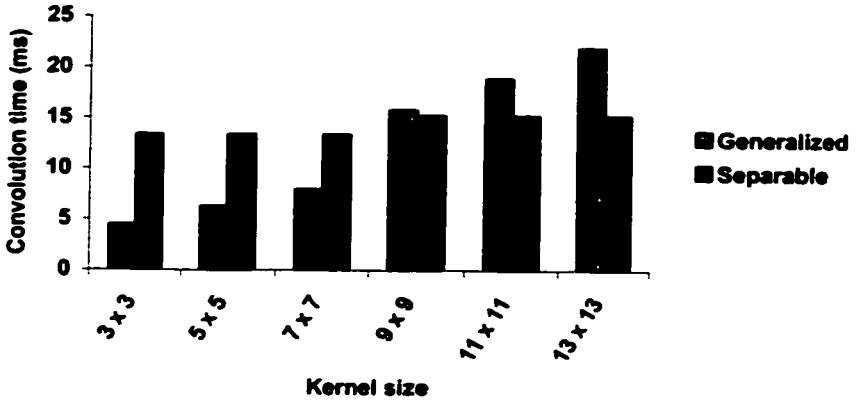


Figure 4-3. Generalized vs. separable convolution on the MAP1000.

Other techniques to reduce the computation include optimization for the specific convolution kernel type, such as the kernels shown in Figure 4-4. If all the kernel elements have a value of 1 (i.e., boxcar kernel, Figure 4-4(a)), then the moving average method can be used to perform convolution [54]. On the MAP1000, we found that due to the availability of *inner-product*, the moving average method and separable convolution method have similar performance. For convolution with the 3 x 3 kernels in Figure 4-4(b) and Figure 4-4(c), we were able to improve the performance by 29% over generalized convolution by eliminating load and multiply/accumulate operations for one entire row of the kernel since they are all zeros. Thus, in general it is possible to improve the

convolution performance for a specific kernel by analyzing the kernel, eliminating loads, skipping other operations or using the kernel's separability property.

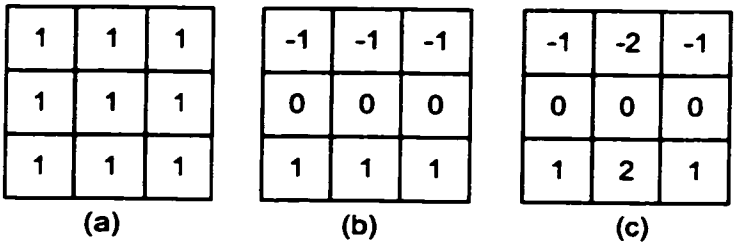


Figure 4-4. Examples of specific convolution kernels.

*4.2 **Median filter***

One of the problems with the averaging filter is that it blurs edges and other sharp details. An alternative is to use a median filter. In echo processing, it is utilized to remove speckles while in color flow processor it is utilized to fill black holes. The echo processor uses a *conventional median filter* while the color flow processor uses an *angular median filter.*

4.2.1 Conventional median filter

Conventional median filtering uses a sorting algorithm to obtain the median value in a given window. There are several sorting algorithms with different efficiency, e.g., the quick sort is a very efficient algorithm. However, most fast sorting algorithms are not suited for small-size windows. To obtain the median value in a 3 x 3 window, the conventional method is to sort the 9 values completely. The procedure is to add each value into a ranked sequence one by one. When the *i*th value is added to the ranked sequence, up to *i-1* comparisons and *i* data movements are needed. Therefore, the maximum number of total comparisons is 8 * (8+1) / 2 = 36 and the maximum number of total data movements is 9 * (9+1) / 2 = 45. Since only the median value is necessary and exact sort sequence is not important, some of the comparisons and data movements can

be omitted. Assume we have sorted 5 pixels. After adding the sixth value to the ranked sequence, only 4 of the 6 values have a chance to be the median value. Thus, when the seventh value is added to the ranked sequence of four candidate values, less comparisons and data movements are needed. This method requires a total of 24 comparisons and 25 data movements at most compared to 36 comparisons and 45 data movements needed by the conventional method.

The MAP1000 has *max_pu8* and *min_pu8* instructions that can be utilized efficiently to perform median filtering. Using these partitioned instructions, eight output pixels can be computed simultaneously by comparing 8 pairs of 8-bit pixels. An example 3-tap median filter without the branching overhead of a sorting algorithm utilizing these instructions is shown in Figure 4-5.

```
/* r0 < r1 */
r0 = min_pu8 (R(x), R(x + 1));
/* r0 = 88 77 60 33 33 47 47 33 */
r1 = max_pu8 (R(x), R(x + 1));
/* r1 = 90 90 77 60 55 55 66 66  */
/*  r0 < r1 <r2 */
r2 = max_pu8 (R(x-1), r0);
/* r2 = 88 88 90 77 60 47 55 33 */
r1 = min_pu8 (r2, r1);
/*median :  r1 = 88 88 77 60 55 47 55 33 */
```

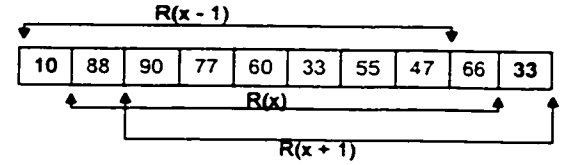


Figure 4-5. A 3-tap median filter.

The non-aligned data, i.e., *R(x - 1)* and *R(x + 1)*, are extracted using *align* instructions from adjacent data words. In our implementation, we extend the above 3-tap filter to a full 3 x 3 median filter. The performance of a 3 x 3 median filter on a 512 x 512 image is 5.85 ms.

4.2.2 Circular median filter

Conventional sorting algorithms on the phase data give incorrect results when the values of flow signals are near aliasing ($\pm \pi$), due to phase wrapping as shown in Figure 4-6. For example, when median filtering with a 3 x 3 data set that contains 5 negative phases (v_l to

v_5) close to aliasing ($-\pi$) and 4 other phases (v_6 to v_9) that have aliased across $-\pi$ becoming positive phases, the least negative phase (v_1) will be incorrectly selected as the median instead of the largest negative phase (v_5). A technique to avoid this artifact is to utilize an angular median filter [55], which would select v_5 as the median value. The angular median can be described as the vector (ϕ), which minimizes the circular mean deviation d:

$$d = \sum_{i=1}^{N} arc_{short}(\phi_i, \phi) \tag{4-2}$$

Eq. (4-2) implies the sum of shortest arc distance of each vector needs to be computed with respect to other vectors, and the vector with the minimum arc distance needs to be picked as the median vector. One method to compute shortest arc distance is:

$$arc_{short}(\phi_i - \phi) = \pi - \left|\pi - \left|\phi_i - \phi_{i+1}\right|\right| \tag{4-3}$$

This computation can be greatly reduced using fixed-point arithmetic:

$$arc_{short} = \left|\ \phi_i - \phi_{i+1}\ \right|$$

and taking advantage of the ability of 2's complement arithmetic to overflow (or wrap around) the number range. This assumes that our measurements for the ϕ data are scaled to the full dynamic range of our signed 2's complement number (e.g., with 8-bit data, maximum red π = 0111 1111 and maximum blue $-\pi$ = 1000 0000). Then, if we compute the signed subtraction of two vectors, $\phi_i - \phi_{i+1}$, ignoring the overflow flag and use the 8-bit result, we are guaranteed to have the shortest arc distance between the two vectors. This method produces a signed result where the sign bit indicates the relative direction of the arc distance.

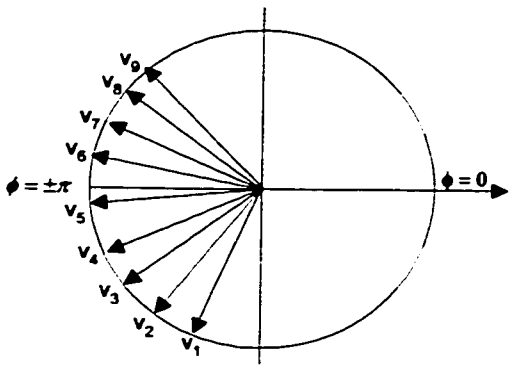


Figure 4-6. Angular vectors.

The shortest arc distance between two vectors on the MAP1000 can be computed by using two instructions: *partitioned_subtract* (to subtract 8 8-bit input pixels) followed by *partitioned_abs* (to obtain absolute of difference). Accumulation of the arc distance can be performed in 16 bits using *partitioned_add* (for 16-bit accumulation, the 8-bit result obtained after subtraction needs to be expanded to 16 bits using *expand*). The accumulation needs to be computed for all 9 pixels. After computing the total arc distance for each vector that has the minimum circular mean deviation can be selected using the *bit_select* instruction discussed in Chapter 3. For computing eight output pixels, the angular median filter requires 312 IFGALU instructions (36 *partitioned_subtract*, 36 *partitioned_abs*, 72 *expand*, 144 partitioned_*add*, 8 *min*, 8 *cle*, 8 *bit_select*) per cluster.

However, for computing the next output pixel, the number of instructions can be reduced by utilizing *distances* and the circular mean deviation d computed for the previous output pixel. For computing the output for the 3 x 3 neighborhood of x_5 (previous neighborhood being centered around x_4), there is only one new incoming column (x_9, x_{12}, x_{15}). Thus, we can subtract the contributions (distances) of the outgoing column (x_0, x_3, x_5) towards the mean deviation d and add the contributions of the incoming column (distances between the incoming column (x_9, x_{12}, x_{15}) and the existing columns (x_1, x_2, x_4) and (x_5, x_7, x_8)) for obtaining the new mean deviation d_{new}. This method eliminates the need to recompute the distances between existing six pixels (x_1, x_2,

x_4, x_5, x_7, x_8). The number of *partitioned_add* instructions is reduced from 144 to 72, but it introduces 36 extra *partitioned_subtracts* (required to subtract the distances of the outgoing column). The number of distances (*partitioned_subtract*) is reduced from 36 to 21, the number of *partitioned_abs* reduced from 36 to 21, and the number of *expand* reduced from 72 to 42. Utilizing this technique, we can compute eight new output pixels in 216 IFGALU instructions instead of 312 IFGALU instructions per cluster. The performance of the 3 x 3 circular median filter on the MAP1000 on a 512 x 512 8-bit image is 32 ms.

4.3 Estimation of phase and magnitude

The phase is estimated by performing *arctangent* on N/D ($\phi = arctan(N/D)$) and magnitude is computed by performing a square root on the sum of squares of N and D ($R = \sqrt{N^2 + D^2}$). Many algorithms have been developed to compute these mathematical functions, of which the CORDIC algorithm and the table-lookup algorithm are prominent. Although the CORDIC algorithm is widely used in hardwired approaches because of simple computations of additions, subtractions and shifts, and the small memory requirement, it is usually not implemented on programmable processors because of the high computational requirement. However, on the MAP1000, where multiple operations for multiple data sets can be executed in each instruction, it is possible for the CORDIC algorithm to obtain good performance. We evaluated the computation of phase and magnitude with both CORDIC and lookup table approach.

4.3.1 CORDIC algorithm

The CORDIC algorithm was first presented by Volder [56] in 1959. An example CORDIC algorithm to compute *square root* and *arctangent* is shown in *Pseudo code p1*, where approximately 1 bit of precision is generated for each iteration of i. If the initial values are $x = D$, $y = N$, and $a = 0$, then after n iterations, x will be $(1/k)(N^2 + D^2)^{1/2}$ represented in n bits, and a will be $arctan(N/D)$ represented in n bits. The scale factor k is equal to 0.60725293.

Pseudo code p1:

```
for i from 0 to n-1 do {
        da = arctan(2^-i);
        dx = x >> i;
        dy = y >> i;
        if (y ≥ 0) s = 1 else s = -1;
        x = x + s dy;
        y = y - s dx;
        a = a + s da;
}
```

The $arctan(2^{-i})$ is computed with a small lookup table with n entries. Since the CORDIC algorithm only involves additions, subtractions and shifts, it can be easily implemented in hardware [57 - 60]. To generate 1 bit of square root and arctangent, the CORDIC algorithm takes 10 operations (1 load, 2 shifts, 1 comparison, 3 conditional negates, 3 add/subtracts). Thus for an 8-bit precision, 80 operations are required to compute both the *square root* and *arctangent*. However, the CORDIC algorithm can be implemented efficiently on the MAP1000 utilizing partitioned operations.

The *partition CORDIC algorithm* utilizing the powerful instructions of the MAP1000 is shown in *Pseudo code p2*. In this *pseudo code*, the data register is 64 bits, the input data are 8 bits, and the number of partitions is eight.

Pseudo code p2:

```
/* Load eight 8-bit N and eight 8-bit D data */
N = *src_u_ptr; D = *src_v_ptr;
/* Take the absolute values of u and v */
x = dif_pu8_ps8(N, 0);  y = dif_pu8_ps8(D, 0);
/* Initialize a to zero */
a = 0;
/* Iterate the loop for n bits of precision */
for i from 0 to n-1 do {
    /*  Right shift x and y by i bits */
    dx = rs_ps8(x, i);
    dy = rs_ps8(y, i);
    /* Load da data */
    da = arctan_LUT(i);
    /* Get the sign of y, and generate a mask */
    sign_y = cge_ps8(y, 0);
    /* Perform both additions and subtractions */
    x1 = add_ps8(x, dy);
    y1 = sub_ps8(y, dx);
```

```
    a1 = add_ps8(a, da);
    x2 = sub_ps8(x, dy);
    y2 = add_ps8(y, dx);
    a2 = sub_ps8(a, da);
    /* Select the result based on the sign of y (mask) */
    x = bit_select(sign_y, x1, x2)
    y = bit_select(sign_y, y1, y2)
    a = bit_select(sign_y, a1, a2)
}
```

To avoid the expensive branch computation, here we perform both additions and subtractions, and use the mask generated by comparing y to zero (*cge_ps8*) to select (*bit_select*) one of those two results.

This *partition CORDIC algorithm* is valid only when both input N and D are positive. Therefore, we take the absolute values of N and D (*dif_pu8_ps8*), before they enter the loop. After the CORDIC computation, the quadrant of the phase (*arctangent*) is determined by checking the signs of N and D using *Pseudo code p3*.

Pseudo code p3:

```
/* Compute π - a */
a1 = sub_ps8(π, a);
/* Get the sign of u */
sign_u = cge_ps8(u, 0);
/* Select a or a1 based on the sign of u */
a2 = bit_select(sign_u, a, a1);
/* Compute 0 - a2 */
a3 = sub_ps8(0, a2);
/* Get the sign of v */
sign_v = cge_ps8(v, 0);
/* Select a2 or a3 based on the sign of v */
a_out = bit_select(sign_v, a2, a3);
/* Store the output magnitude and phase */
*dst_mag_ptr = x;  *dst_phase_ptr = a_out;
```

```
[ /* "C" pseudo code explaining
     the quadrant determination */
     α1 = π - α;
     if (u ≥ 0) α2 = α else α2 = α1;
     α3 = - α2;
     if (v ≥ 0) α_out = α2 else α_out = α3;]
```

The *Pseudo code p2* and the *Pseudo code p3* of the *partition CORDIC algorithm* are for 8-bit data. For 16-bit data, the MAP1000 has another set of instructions with 16-bit partitions.

4.3.2 Lookup table

In the table-lookup algorithm, values of the mathematical function (*f(x)*) are precomputed and stored in a lookup table (LUT). The argument x is an index to the LUT. Since the LUT is precomputed, it can be fixed to the desired level of precision, i.e., the number of bits in each LUT entry.

To perform *square root* ($(N^2 + D^2)^{1/2}$) and *arctangent* (*arctan(N/D)*) using the table-lookup algorithm, one method is to concatenate the two arguments N and D to form a single index into the LUT. This method requires a large LUT, e.g., for 12-bit N and 12-bit D, an *arctangent* LUT with 16 bits per entry requires 32 Mbytes of memory. Since the on-chip memory is relatively small, the large LUT has to reside in the slow off-chip memory, which causes a large performance penalty. To avoid a large LUT, a hybrid table-lookup algorithm can be used, in which the *square root* is computed in the floating-point unit and *arctangent* is performed by table lookup.

In this hybrid table-lookup algorithm, the partitioned floating-point square root instruction can be used to compute *square root* ($(N^2 + D^2)^{1/2}$). Since the *arctangent* LUT has a disadvantage of an infinite index ($tan(\pi/2) = \infty$), which means a poor accuracy for a LUT, we perform $D/(N^2 + D^2)^{1/2}$ (we already have the value of $(N^2 + D^2)^{1/2}$), which is always less than or equal to one, and then use an *arccosine* LUT to get the phase. Only a LUT for one quadrant is nessesary, since the values for the other quadrants can be determined by using *Pseudo code p3*.

On the MAP1000, the LUT operation can be handled either by the data cache or by the on-chip programmable DMA controller. In the data cache mechanism, if the needed data do not reside in the cache, a cache miss occurs, incurring the overhead from fetching the data to cache. Since the data cache is optimized for sequential memory access, if the addresses of the input data are not sequential, which is typical, the data cache-based method will cause a large number of cache misses. In addition, as the LUT size is increased compared to the limited data cache size, the probability of data cache misses further increases. The MAP1000 DMA controller's *guided transfer*, discussed in Chapter 3, can also be used to perform table-lookup operations. The core processor

performs the magnitude and $(D/\sqrt{N^2+D^2})$ computation while the DMA controller implements the *arccosine* LUT using $(D/\sqrt{N^2+D^2})$ as an index. This method still has the overhead of the SDRAM row access penalties, but minimizes the penalty of the cache misses. Similar technique is used in the echo processor while implementing the log_LUT on the magnitude computed $\sqrt{I^2+Q^2}$ data [61].

4.3.3 Performance comparison between CORDIC and lookup table

The performance on 512 × 512 8-bit random input data is plotted in Figure 4-7. The data choosen are random so that the performance obtained with LUT represents the worst case scenario. For the CORDIC algorithm, to get the output data of *n*-bit precision, *n* iterations are needed. Thus when the precision of the output data increases from 5 bits to 8 bits, or from 9 bits to 16 bits, the execution time increases approximately linearly. The large jump in the execution time when the input data size (bits) increases beyond 8 is mainly due to the change in the number of partitions in a register. As shown in Figure 4-7, when the precision of the output data is less than 11 bits, the *partition CORDIC algorithm* offer better performance than the table-lookup algorithms.

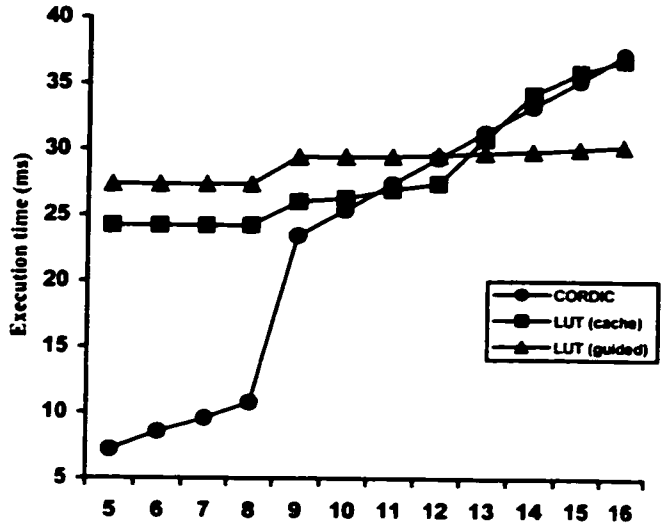


Figure 4-7. Comparison in performance between the partition CORDIC algorithm and the table lookup algorithms.

For the cache-based table-lookup algorithm, the LUT size increases with the increase in the output precision, incurring more data cache misses due to the limited data cache size. Thus the execution time increases when the output precision increases. For the guided transfer-based table-lookup algorithm, data in the LUT are loaded individually from off-chip memory by the DMA controller whether they are already in the on-chip data cache or not. Thus there is no significant change in the performance when the precision (or LUT size) increases. However, with the increase in the precision, the execution time increases slightly due to the overhead from the increased SDRAM row misses. The jump in the execution time when the input data size increases beyond 8 is mainly due to the change in the number of partitions in a register for computing *square root*. As shown in Figure 4-7, when the precision of the output data is larger than 12 bits, the guided transfer-based table-lookup algorithm can perform better than the cache-based table-lookup algorithm. Thus, depending upon the precsion, size of LUT and performance requirement, any one of these three methods can be selected to perform *arctangent* and *square root*.

4.4 Wall filter

The typical wall filtering techniques to remove clutter signals include a finite impulse response filter (FIR), an infinite impulse response filter (IIR) and an autoregressive filter. In the following section, we will discuss the implementation of each of these techniques on the MAP1000.

Since the ensemble size (E) varies from 4 to 16, it is not always guaranteed that the adjacent vectors in the input data are aligned on the 64-bit boundary. For example, when E is equal to 4, 8 or 16, then adjacent vectors are aligned. When $E = 6$, every second vector is aligned, and when $E = 7$, every fourth vector is aligned. Thus, the alignment location depends upon the ensemble size. This unaligned data accesses and arbitrary alignments introduce an overhead in computation as well as complexity in implementation. Thus, careful handling of these unaligned data accesses is essential in implementing an algorithm that has E as the input parameter, e.g., *autocorrelation*, *wall filter*, *clutter_shift*, etc. One simple and effective solution is to preprocess the data and insert dummy ensembles so that adjacent vectors get aligned on the 64-bit boundary. For example, when $E = 6$, insert two dummy ensembles, when $E = 7$ insert 1 dummy ensemble, and when $E = 9$ insert 3 dummy ensembles.

4.4.1 FIR wall filter

For an FIR filter to achieve the desired level of rejection, the number of taps should be large, and the cutoff frequency may be required to be higher than the desired cutoff to more completely remove the clutter from the signal. However, this has an adverse effect of removing some of the Doppler signal. An FIR filter with m taps is described by the following equation:

$$y_n = \sum_{i=0}^{m-1} a_i x_{n-i} \tag{4-4}$$

where a_i is the ith filter coefficient, x represents an input signal and y is the filtered output signal. The FIR tap size normally used for a wall filter ranges from 3 to 5. One of the disadvantages of using an FIR filter is a reduction of the number of data samples at their

output. This means that after the application of an FIR filter, there are even fewer samples left in the ensemble available from which to estimate the Doppler frequency, e.g., if $E = 12$ and $m = 5$, then the number of samples left over after the FIR-based wall filter would be only 8 (12 – 5 + 1).

The *inner-product* instruction utilized for convolution can be used for the FIR-based wall filter. Since the tap size is always less than eight, every execution of this instruction yields one wall-filtered output. We keep executing the instruction throughout the image by right shifting in a new pixel. Invalid pixels are generated when all the filter taps do not lie within a single vector, i.e., when some filter coefficients are multiplying with the data belonging to the previous vector and the other filter coefficients are multiplying with the data belonging to the current vector. The Data Streamer is programmed to discard these invalid pixels while storing the filtered data to the memory. Thus, we do not have to consider the vector or ensemble boundaries. We implemented this method on the MAP1000 and obtained a performance of 16.3 ms for a 256 x 512 image when the ensemble size is 6.

4.4.2 IIR wall filter

IIR filters are recursive filters whose response to an impulse input is infinite in time. The output at a given instant is a linear combination of previous input and output values. IIR filters improve the rejection of the unwanted clutter and allow for a narrower transition band between the rejected and unaffected velocities. An *m*th-order IIR filter is described by the equation:

$$y_n = \sum_{i=1}^{m} b_{i-1} y_{n-i} + \sum_{i=0}^{p-1} a_i x_{n-i} \qquad (4\text{-}5)$$

where y is the filtered output signal, a and b are the filter coefficients, x is the input signal, and p is the number of taps on the input signal. The order of the IIR-based wall filter is normally 2 ($m = 2, p = 3$).

Implementing an IIR filter efficiently is a challenge due to the dependency of the current output on the previous output. Even though we can implement Eq. (4-6) using

inner-product, the new output has to wait until all the latency cycles are satisfied for the previous output, i.e., 6 on the MAP-CA, which limits software pipelining. The next challenge is, while utilizing *inner-product*, y and x need to be in the same 64-bit register, i.e., y and x need to be interleaved. Thus, additional instructions, such as *extract* and *shift* type of operations, are necessary to combine x and previously-computed y before the *inner-product* instruction can be utilized to compute the new output pixel. In addition, the powerful auto-shift property cannot be used since two new pixels (x_{new} and $y_{previous}$) need to be shifted in from two different sources. To overcome these difficulties, we have reordered the IIR filter computation in Eq. (4-6) so that only input pixels are utilized to compute the new output pixels independent of the previous output pixels. Our reordering is better illustrated with an example as shown below for the IIR order of 1, the FIR order of 2 and the ensemble size of 6:

$$y_0 = b_0 y_{-1} + a_1 x_{-1} + a_0 x_0$$

$$y_1 = b_0 y_0 + a_1 x_0 + a_0 x_1 = b_0(b_0 y_{-1} + a_1 x_{-1} + a_0 x_0) + a_1 x_0 + a_0 x_1$$

$$= b_0^2 y_{-1} + b_0 a_1 x_{-1} + b_0 a_0 x_0 + a_1 x_0 + a_0 x_1 = b_0^2 y_{-1} + b_0 a_1 x_{-1} + A x_0 + a_0 x_1$$

$$y_2 = b_0 y_1 + a_1 x_1 + a_0 x_2 = b_0^3 y_{-1} + b_0^2 a_1 x_{-1} + b_0 A x_0 + A x_1 + a_0 x_2$$

$$y_3 = b_0 y_2 + a_1 x_2 + a_0 x_3 = b_0^4 y_{-1} + b_0^3 a_1 x_{-1} + b_0^2 A x_0 + b_0 A x_1 + A x_2 + a_0 x_3$$

$$y_4 = b_0 y_3 + a_1 x_3 + a_0 x_4 = b_0^5 y_{-1} + b_0^4 a_1 x_{-1} + b_0^3 A x_0 + b_0^2 A x_1 + b_0 A x_2 + A x_3 + a_0 x_4$$

$$y_5 = b_0^6 y_{-1} + b_0^5 a_1 x_{-1} + b_0^4 A x_0 + b_0^3 A x_1 + b_0^2 A x_2 + A b_0 x_3 + A x_4 + a_0 x_5$$

where x_{-1} and y_{-1} are the initial conditions of input and output signals controlling the transient response of the IIR filter. One simple approach is the step initialization where all initial parameters are set to zero.

The above IIR filter expansions have the following features, which can be utilized for efficient implementation:

(1) There is no interdependency between two consecutive outputs and all the outputs depend upon the input signal values and initial conditions. Thus, software pipelining can be easily performed and extra instructions, such as *extract* and *shift* type of operations, are not necessary.

(2) As we progress in computation of y (e.g., y_1, y_2, y_3, y_4 and y_5), x_0 gets multiplied with the new coefficient while the rest of input signal value (e.g., x_1, x_2, x_3, x_4 and x_5) gets multiplied with the preceding coefficients. For example, while computing y_2, x_0 gets multiplied with b_0A, x_1 gets multiplied with A, x_2 gets multiplied with a_0, and while computing y_3, x_0 gets multiplied with the new coefficient b_0^2A while x_1 gets multiplied with b_0A (which was previously the coefficient for x_0), x_2 gets multiplied with A (which was previously the coefficient for x_1) and x_3 gets multiplied with a_0 (which was previously the coefficient for x_2). Thus, the auto-shift property of *inner-product-add* can be efficiently utilized. In contrast to the FIR implementation where PLC has the FIR coefficients and PLV has the input pixels, PLC has the input pixels and PLV has the kernel coefficients for this IIR implementation.

(3) y_{-1} and x_{-1} also get multiplied with the new coefficient for each output pixel. By pre-computing these two terms and setting this register as the destination register while performing *inner-product-add*, we can efficiently compute the IIR wall filter.

Utilizing this method, we implemented IIR on the MAP1000 and obtained a performance of 38.1 ms for a 256 x 512 image when the ensemble size is 6.

4.4.3 Autoregressive filter

Regression wall filters treat their inputs as polynomial functions in time domain and operate on the assumption that the slowly-varying clutter component can be approximated by a polynomial of a given order. Once approximated, this component can then be subtracted from the original signal so that the contribution from the blood flow can be retrieved and analyzed. Mathematically, it can be described by

$$y(k) = x(k) - \sum_{d=0}^{D} a_d k^d \quad k = 1, 2, \ldots E \tag{4-6}$$

where x and y are input and output signals at ensemble k, a_d is a set of regression model coefficients, D is the regression order, and E is the ensemble size. To perform the least-squares analysis, Kadi and Loupas [28] suggested the use of an $E \times D$ Vandermonde matrix,

$$V = \begin{bmatrix} 1^0 & 1^1 & \cdots & 1^D \\ 2^0 & 2^1 & \cdots & 2^D \\ \vdots & \vdots & \ddots & \vdots \\ E^0 & E^1 & \cdots & E^D \end{bmatrix} \tag{4-7}$$

to generate the coefficients for the clutter approximating polynomial, a_d. The clutter approximating polynomial is simply VMX, where M is a $D \times E$ matrix given by

$$M = (V^T V)^{-1} V^T \tag{4-8}$$

and X is the input ensemble $E \times 1$ vector. Then the filtered data are given by,

$$Y = X - VMX \tag{4-9}$$

Since the ensemble size and order D is known beforehand, VM can be pre-computed and stored in a lookup table to reduce the computations significantly. So, the wall filter computation is reduced to a matrix multiplication.

The MAP1000's partitioned *inner-product* instruction that performs eight 16-bit multiplications and accumulation in a single instruction is very useful. When E = 16, two *inner-products* can be utilized to compute one *wall filter* output pixel. We implemented autoregresssion efficiently on the MAP1000 using *inner-product* instructions. Our implementation of an autoregression wall filter on the MAP1000 takes 20.7 ms for a 256 x 512 image when $E = 6$.

The performance listed for FIR and IIR was obtained after implementing the algorithms in C with intrinsics. However, we implemented autoregression algorithm in assembly since it is more effective in removing clutter signals compared to other filters [28] and the optimum performance was needed in minimizing the number of processors required to implement the system.

4.5 First lag of autocorrelation and clutter_shift

The computation of the first lag of *autocorrelation* requires complex multiplications between the n-element ensemble and the shifted version of itself. There are three main steps in computing autocorrelation; (1) obtain the shifted version of itself; (2) obtain the complex conjugate of the shifted version; and (3) perform complex multiplications between the original signal and the shifted version (complex conjugate) for the ensemble length as shown in Eqs. (2-6) and (2-7).

The MAP1000 has a *complex_multiply* instruction, shown in Figure 3-5, which can perform multiplications and additions for 2 sets of complex numbers in a single instruction. This instruction is very useful in computing the third step. However, I and Q data need to be interleaved with each other to utilize this instruction. *shuffle* can perform the interleave operation. The shifted version of itself is obtained by using a simple *move.32* instruction from the interleaved IQ data since the interleaved IQ data are 32-bit aligned and *move.32* operates on 32-bit operands. The complex conjugate is obtained by multiplying Q data with −1 using *mpy.ps16* that can perform four 16-bit partitioned multiplications in a single instruction. *autocorrelation* also has the unaligned data access problem similar to the wall filter. Thus, the data can be pre-formatted so that vectors are always 64-bit aligned. We implemented *autocorrelation* utilizing these instructions and obtained a performance of 11.0 ms for a 256 x 512 x 6 image. The compute time is only 5.7 ms while the time required to bring the data on-chip and write the result off-chip is 8.4 ms. Thus, it is an I/O-bound function.

The computation of *clutter shift* utilized to shift the clutter is similar to that of *autocorrelation* and requires a complex multiplication:

$$(I(k) + jQ(k))(\cos\omega_c k + j\sin\omega_c k) \qquad (4\text{-}10)$$

shuffle can be utilized to interleave I and Q here as well. *cosine* and *sine* functions can be precomputed and stored in a LUT. *clutter_shift* on the MAP1000 for a 256 x 512 x 6 image takes 16.5 ms. The compute time is 5.5 ms while I/O time is 15.6 ms. Thus *clutter_shift* is also an I/O-bound function.

4.6 Scan conversion, frame interpolation and tissue flow decision

Scan conversion transforms the acquired polar coordinate data to the geometry and scale of the sector scan on the Cartesian raster output image. The key tasks for scan conversion are (1) to calculate the address of the input data $V(\varphi,r)$ (i.e., a polar conversion, requiring $\varphi = arctan(y/x)$ and $r = \sqrt{x^2 + y^2}$) and the interpolation coefficient weights for each output pixel $P(x, y)$; (2) to fetch the neighboring input data values for a 4 x 2 interpolation; and (3) to compute the 4 x 2 interpolation. For step (1) a pre-computed lookup table can be used since the geometry is known beforehand and for step (3) *inner-product* can be utilized. The main bottleneck in scan conversion is step (2), i.e., fetching the input data. With the input address lookup table, there are three approaches in accessing the input pixel groups: cache-based, DMA-based guided transfer, and DMA-based 2D block transfer. In the cache-based implementation, the core processor directly loads the input data, while in the DMA-based guided transfer the DMA controller directly brings the 4 x 2 group of input pixels for each output pixel. In the DMA-based 2D block transfer, a large 2D block of input data corresponding to a 2D block of output pixels is brought on-chip. We experimented with these three methods, finding that they took 49.6, 37.1 and 23.8 ms, respectively, for the computation of an 800 x 600 output image with an input image of 512 x 1024 [61]. Frame interpolation utilizes the subword parallelism of the MAP1000, using the partitioned *multiply*, *add*, and *subtract* instructions. The tissue/flow algorithm above is a classic *if/then/else* algorithm, which is remapped using techniques discussed in Section 3.3.4 Scan conversion, frame interpolation and tissue flow decisions are discussed in detail by York [61].

4.7 Miscellaneous functions

Other miscellaneous functions include *persistence* and *croner_turn* (*transpose*). For an efficient *corner_turn*, both the DMA controller and core processor are used. The DMA brings in 2D data blocks similar to Figure 3-15 and then outputs the 2D blocks in transposed order. Meanwhile, the core processor uses special partitioned operations, e.g., *shuffle* and *combine*, to transpose the 2D block as shown in Figure 3-3. *persistence* requires multiply-add type of operations, which can be implemented efficiently using the inner-product instructions. Both *croner_turn* and *persistence* are I/O bound functions. For a 256 x 512 x 6 input image (for both I and Q image) *corner_turn* takes 29.2 ms of which only 4.7 ms is compute time and for a 512 x 1024 image *persistence* takes 12.9 ms of which only 5.9 ms is compute time.

By efficiently mapping the algorithms, we were able to reduce the processing requirements significantly. In the next chapter, we summarize the performance of each algorithm, present multiprocessor architectures and finally show our multiprocessor simulation results.

Chapter 5: Multiprocessor Architecture for Color Flow System

In this chapter first we present the overall single processor simulation results. Based on these results, we present our multi-mediaprocessor architecture for ultrasound processing. This chapter concludes with the results of our B-mode and color-mode simulations on this architecture, demonstrating that an ultrasound machine with programmable mediaprocessors is feasible.

5.1 ***Single processor simulation results***

Many of the algorithms discussed in Chapter 4 are I/O-bound i.e., the time required to bring the data on-chip and store the processed data off-chip ($t_{i/o}$) is more than the computation time $t_{compute}$. If each algorithm is implemented individually, requiring the data to be moved between on and off chip for each algorithm, then the total estimated time is $t_{total} = \sum_{j} \max(t_{compute_j}, t_{i/o_i})$ (where j stands for each algorithm). Thus, if the algorithms are implemented individually the total time required to implement all the algorithms would be 331.27 ms. However, by combining the tight-loops of several I/O-bound functions and sharing the data, it is possible to reduce the total time required.

While sharing the data, once a block of data is brought on-chip for the first routine, it is continually processed by the subsequent routines as much as possible before storing the results off-chip. The total time using data sharing would then be $t_{total} = \max((\sum_{j} t_{compute_j}), t_{i/o-shared})$. *autocorrelation* ($t_{i/o}$ is 8.4 ms and $t_{compute}$ is 5.7 ms), *clutter-shift* ($t_{i/o}$ is 14.6 ms and $t_{compute}$ is 4.3 ms), *corner-turn* ($t_{i/o}$ is 14.4 ms and $t_{compute}$ is 4.72 ms) are several examples of I/O-bound functions. Thus by combining these algorithms efficiently with other algorithms to share the I/O, it is possible to reduce the total time. A flow graph of combining *autocorrelation*, *clutter-shift* and *corner-turn*

algorithms with the computation of *arctangent*, *wall filter* and *clutter-filter* algorithms is shown in Figure 5-1. A *combineI&Q* function interleaves I and Q data required for the autocorrelation. Once I and Q are interleaved, *autocorrelation* tight loop can be executed on the IQ and IQ^* data which are already present in the cache. *autocorrelation* generates separate N and D image data, which are processed using 1D FIR filters to remove noise. I and Q data which were utilized by *combine_I&Q* function, and phase generated by frequency estimators will be utilized by *clutter-shift* function to shift the clutter so that it gets centered around zero frequency. This interleaved IQ data is again de-interleaved into I and Q to facilitate wall filter. The *autocorrelation* function can then again process the filtered I and Q to generate N and D.

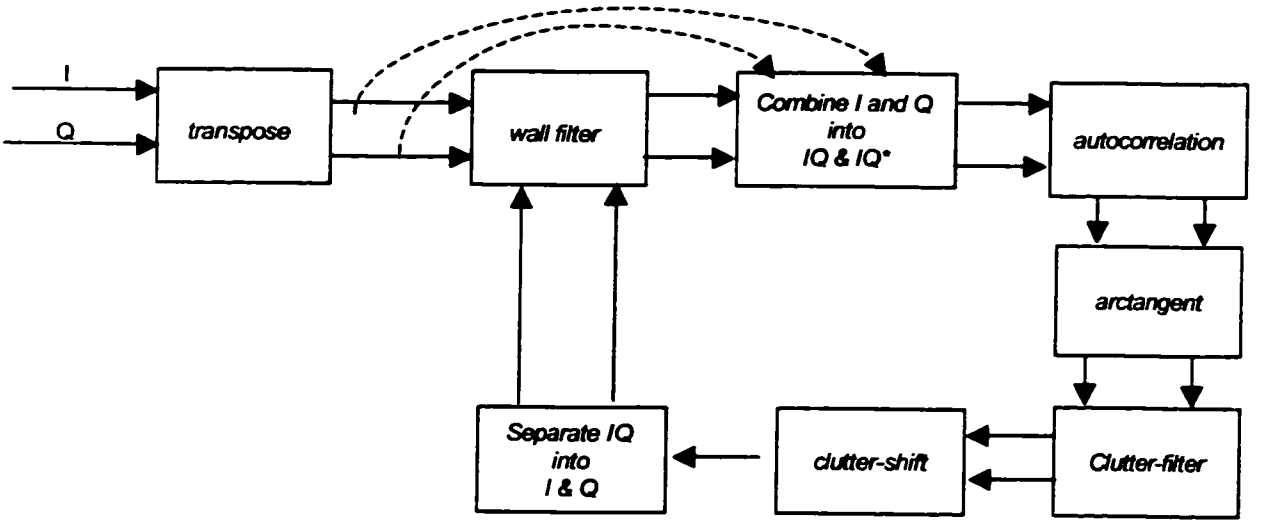


Figure 5-1. Sharing of I/O between different algorithms in *CF part 1*.

This whole process is performed using *in place* replacement (multiple usage of cache memory by different algorithms) of cache thus utilizing the limited on-chip memory efficiently. Similarly, algorithms in *CF part 2* and *EP* are combined to make several I/O-bounds functions into compute-bound functions. Table 5-1 lists the performance of all algorithms after sharing the data flow. The total performance now reduces to 229.4 ms thus saving 35% of computation time.

Table 5-1. Performance of combined algorithms

Processing modules	Algorithms (time in ms)	Performance		
		Code	$t_{compute}$	Total
Color flow part1 256 x 512 x 6	Corner_turn (transpose)	C	4.72	5.31
	Autocorrelation	C	12.74	15.26
	Separate (N&D and I&Q)	C	5.26	6.96
	ϕ and R, 5-bit CORDIC	C	4.53	5.62
	Clutter_shift	C	5.51	8.76
	2D FIR	Asm	2.00	3.24
	Wall_filter	Asm	18.85	20.65
	Total			**78.43**
Color flow part2 256 x 512	2D FIR (3x3)	ASM	3.53	4.48
	ϕ and R, 8-bit CORDIC	C	5.32	5.95
	Key hole filter	C	0.45	0.65
	Median filter (3x3)	C	7.42	7.95
	Persistence	C	2.66	5.67
	Total			**28.91**
Echo processor 512 x 1024	Magnitude and log compress	C	16.40	34.40
	2D FIR	Asm	10.3	12.4
	Black_hole_fill	C	5.1	6.5
	Corner_turn	C	1.7	1.9
	Persistence	C	6.5	8.4
	Total			**63.5**
SC,FI and TFD 800 x 600	SC Color-mode	C	24.2	28.652
	SC B-mode	C	12.90	23.80
	FI and TFD	C	2.9	6.1
	Total			**58.6**

5.2 Overall results of ultrasound algorithm mapping

The results of mapping and simulating the various ultrasound algorithm stages (echo processing, EP; color flow, CF; scan conversion, SC; frame interpolation, FI and tissue/flow decision, TFD) for all the specified scenarios from Chapter 2 are shown in

Table 5-2. Estimated number of MAP1000 processors needed for various scenarios.

Mode	k	C fps	B fps	# C vectors	# B vectors	E	ROI	sector angle	Number of Map1000s EP	CF	SC/FI/TF	Total
B			68.0		512			136	5.17		1.42	6.74
B			68.0		340			90	3.44		1.73	5.16
color	1	9.0	9.0	256	340	16	100%	90	0.46	2.42	0.66	3.54
	2	8.4	16.8						0.85	2.25	0.89	3.99
	3	7.8	23.5						1.19	2.10	1.10	4.38
	4	7.3	29.3						1.48	1.97	1.27	4.73
	5	6.9	34.5						1.74	1.85	1.43	5.03
color	1	22.3	22.3	256	256	6	100%	90	0.85	2.62	1.61	5.08
	2	19.5	39.1						1.49	2.29	2.07	5.85
	3	17.4	52.1						1.98	2.04	2.42	6.44
	4	15.6	62.5						2.38	1.83	2.70	6.91
color		68.0	68.0	52	256	6	20%	90	2.59	1.62	2.23	6.44

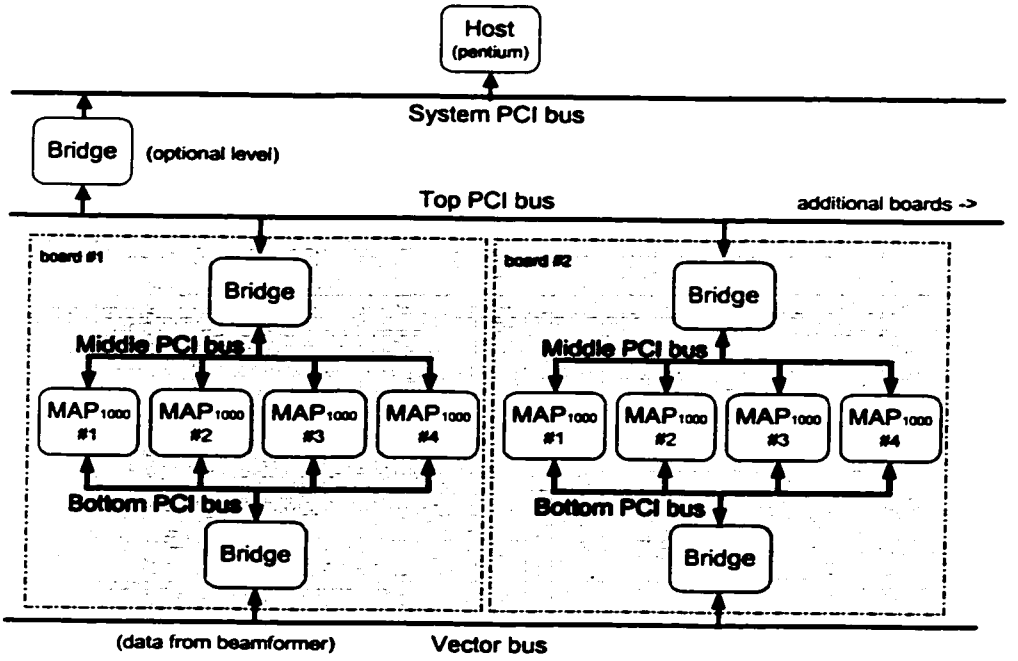
Table 5-2. The number of MAP1000 processors required to meet the frame rate requirement for a particular scenario is obtained by multiplying the total time required to process one frame on individual MAP1000 with the total frame rate requirement (frames/second). The total execution time includes not only the execution time for each stage, but an additional I/O time ($t_{i/o}$) due to incoming data frames from the previous processing stage. These incoming frames are double-buffered. Thus, some of this $t_{i/o}$ may be hidden behind the computation of the respective stage. Of the above scenarios, the worst case B-mode and color-mode simulations require about seven MAP1000s, with 6.74 and 6.91, respectively. Based on these initial results, we designed our architecture based on eight MAP1000s, but expandable to sixteen MAP1000s.

5.3 Multiprocessor architecture

Our goal in designing a multiprocessor architecture is not only to meet the high computational requirements of ultrasound processing, but to arrive at a cost-effective system as well. Our approach to meet this goal was to design one board that could be replicated, scaling to the processing requirements. We used a standard bus structure and standard memory components with a minimum interface logic to the mediaprocessor. Figure 5-2 shows a block diagram of our architecture, designed to handle the ultrasound processing stages illustrated in Figure 1-2 following the RF demodulator stage. The board is composed of 4 MAP1000 processors, where each processor node (PN) has its own local 64-Mbyte SDRAM and two PCI bus ports.

Our architecture meets the data flow requirements of raw and intermediate ultrasound data. The data vectors arrive from one end of the board (vector bus). After a series of pipelined processing stages, the data are sent out the other end of the board to the host PC for display. In order to make the maximum use of each processor, it is critical that each processor wait as little as possible for data and that interprocessor communication be efficient. The input data vectors from the RF demodulator are received from the vector bus by a bridge logic that routes the vectors to the appropriate processor's local memory through one of its PCI buses. Double buffering is used such that while the MAP1000 is processing the current image frame stored in one local buffer, the beamformer is storing the next image frame in another local buffer. Since the data transfer time is less than the compute time, the MAP1000s with double buffering effectively do not have to wait for data, and the throughput is determined by the processing load on each MAP1000. However, the MAP1000's processing time can be impacted by the incoming beamformer vector bus traffic that must compete with the Map1000's Data Streamer for access to local memory. The Data Streamer is programmed to transfer the data between local memory and on-chip memory as well as transfer the processed frame to the next MAP1000 processor's local memory through the PCI bus for the next stage of pipelined processing (again using double buffering). The overhead of this internal bus traffic within the MAP1000 is not significant due to the presence of two

high-speed internal buses (3.2 GBps). After the data are processed, they are sent though



the system PCI bus to the host PC, which handles the graphical user interface and displaying the final processed images.

Figure 5-2. UWGSP10 architecture utilizing 2 PCI ports per MAP1000 processor.

In case we can use only a single PCI port on the MAP1000, Figure 5-3 shows the architecture using MAP1000s utilizing one PCI port. Both the single and dual-PCI port architectures were simulated in this study.

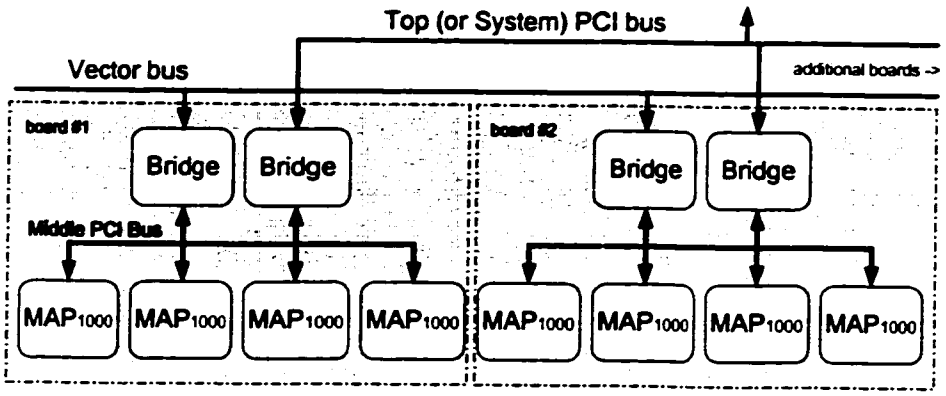


Figure 5-3. UWGSP10 architecture utilizing 1 PCI port per MAP1000 processor.

For our primary architectures, the interprocessor communication is based on a simple message passing paradigm [33], in which the destination processor is assumed to be *polling* certain predefined memory addresses (either continually or at regular intervals) waiting on a message (32 bytes) from the source. The source processor then sends the message (using a normal PCI write) to the destination processor's memory. The destination processor then returns an acknowledge message.

Each MAP1000 processor can support up to 64 Mbytes of local SDRAM memory for a total of 512 Mbytes for a 2-board (8-processor) system. Of the 64 Mbytes of SDRAM per MAP1000, only about 10 Mbytes are used for program memory and buffer space leaving the rest for CINE memory. CINE memory is used to store few frames of data, which can be later played back in real time by the clinician. With the fairly large amount of memory free at each processing node, we can store the CINE data before the filter stage (after *EP part 1* for B-mode and after *CF part 1* for color-mode). Since we can store the CINE loop data after any stage of processing, by choosing to store after *EP part 1* and *CF part 1* allows all the filters to be modified by the user during CINE playback, e.g., changing persistence, the degree of speckle reduction or edge

enhancement, zoom or rotation of scan conversion, and the thresholds for tissue/flow decision.

In addition to ensuring that our system can meet the processing requirements, it must have adequate bus bandwidth to handle the high amount of data flow in ultrasound processing. Even though the 64-bit PCI bus at 33 MHz and the 32-bit PCI bus at 66 MHz appear to have the same peak bandwidth (BW_{max}) of 264 Mbytes/s (MBps), not all of this bandwidth is available to transfer data, since some bandwidth is used in overhead cycles. The effective bandwidth available for data can be estimated from:

$$BW_{eff} = \frac{data\ cycles}{address\ cycles + data\ cycles + spin\ cycles} \cdot BW_{max} \qquad (5\text{-}1)$$

where *address cycles* are the cycles utilized for the address transfer, *data cycles* are the cycles utilized for transferring the data, and *spin cycles* are the cycles needed for the processor to release the bus and the other processor to acquire the bus.

This BW_{eff} estimate does not include any overhead due to bus arbitration and does not factor in *back-to-back* transfers on the PCI bus,(in which the current processor can send back-to-back transfers avoiding the overhead of the spin cycles when there is no second processor arbitrating for the bus). Similarly the estimate in Table 5-2 also does not account for any overhead that would be experienced in typical multiprocessor architectures such as interprocessor communication delays, bus traffic and contention between processors. In addition, division of the processing load across parallel processors incur additional load if the processing load is not evenly balanced between stages (due to different grain size of the processing tasks) and due to overlapped data vectors being recomputed between parallel processors. Thus in order to estimate the actual processing and bus load multiprocessor simulation is essential.

*5.4 **Multiprocessor simulation.***

The goal of the multiprocessor simulation is to evaluate if there is enough bus bandwidth between the processors and what impact the interprocessor communication and bus

traffic have on the overall processing performance. A common approach to study the bus load is to record the detailed timing of all the reads and writes across the bus, known as *address traces* when running the application algorithms for each processor in the system. Various methods have been developed by different researchers for the collection of address traces [63]. We have used simulation-based traces, where the real hardware is modeled in software including the memory, instruction fetch and ALU operations and the required address trace is collected from this simulator. The software simulation models can be either behavioral or structural [65]. Structural models mimic the internal logic gates and electrical timing of a device and tend to be very accurate while the behavioral-level models tend to be cycle-accurate, modeling the device behavior at a higher abstraction via high-level programming languages or hardware description languages like VHDL. For multiprocessor simulations, the behavioral model approach combined with address tracing can lead to reasonable accuracy with a manageable simulation time. Thus, two major steps for behavioral model simulation to obtain processor and bus load in the multiprocessor environment are to generate address trace file and then run them on the VHDL models simulating the system.

5.4.1 Address trace generation

In Chapter 4, we simulated the ultrasound algorithms targeted to a single processor using Equator's MAP1000 cycle-accurate simulator, CASIM. CASIM is designed to model the complex interaction between the core processor, the 4-way set-associative data cache with 4 banks, 2 way set-associative instruction cache, DMA controller, and SDRAM. It generates detailed instruction and data flow timing information in *memory.trace* file. CASIM's accuracy is +/-5% of the real hardware. For our multiprocessor simulation, ideally we would like to run multiple CASIM simulations for each processor, having the programs dynamically interact cycle by cycle. However, we cannot use CASIM directly for the multiprocessor simulation, as it produces static address traces (*memory.trace*) instead of allowing dynamic interaction. In addition, CASIM does not simulate the PCI ports needed for the multiprocessor simulation. We created a simulation process where

we utilized CASIM to generate address trace file and then utilized VHDL model for behavioral model multiprocessor architectures. Our simulation process is shown in Figure 5-4 with the following features to maintain accuracy while reducing the simulation complexity and time.

1. Run the compiled code on CASIM.
2. CASIM generates extreme details of data access patterns, as shown in Appendix B, creating a large file. For example, a simple convolution of a 512 x 512 image with a 3 x 3 kernel with 4.5 ms of execution time creates a 420-Mbyte debug file. We developed a *parser* program to extract only critical information from the *memory.trace* file as it is created during the CASIM simulation before being stored, resulting in a 1200-to-1 reduction in the final ATF file size. For example, the convolution final ATF is around 350 kbytes.
3. Generate two ATF files: one for the core processor and other for the Data Streamer (DS). For the core processor ATF, we model cache line read/writes using *rb/wb* and computation time as address-no-operation (*an)* mnemonics. The core processor also controls the DS using key events such as *DsKick* and *DsWait* by using mnemonics *is* and *wt* respectively. The ATF of the data streamer contain the necessary details for each I/O transfer (the memory addresses, the block width, block height, and the DS mode such as stop, or continue).
4. The control signals to communicate across the processors are embedded within the address trace file. For example, frame end signals are used to indicate whether the transfer of frame from one processor to other processor is complete.
5. The address trace file generated for individual processors are then remapped to the multiple processors by dividing the algorithms across several processors and updating the address filed of the memory transaction.

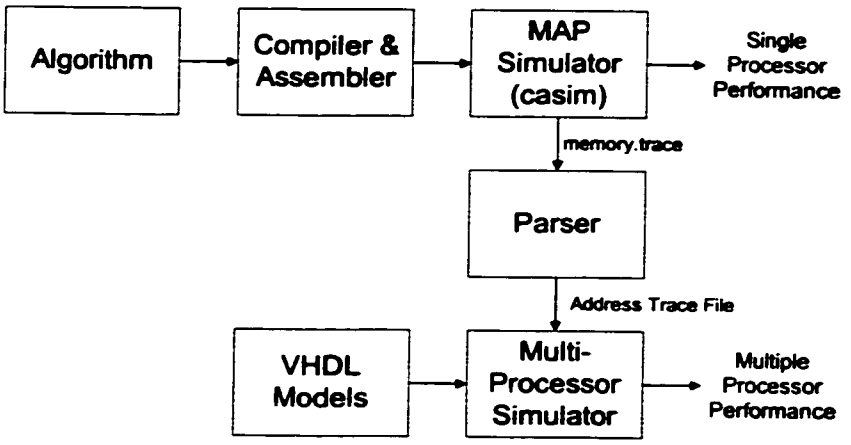


Figure 5-4. Address trace generation process using CASIM.

The example address trace file for processor and data streamer along with the control signals is shown below.

```
For the processor ATF
        an 14          -- anop <cycles>
        rb 80000 4     -- read  <addr> <burst_size>
        wb 90000 4     -- write  <addr> <burst_size>
        is #           -- DsKick DMA transfer <channel #>
        an 467               -- anop
        wt #1          -- DsWait on DMA transfer <channel
        #1>
        fr 0                 -- Wait for the frame signal from a
        different
                             processor
For the data streamer
        r1 90000  64      16     448     1    -- 2D_read
          <addr><width><count><pitch><halt>
```

VHDL models are then developed to run the generated address trace file and thus simulate the multiprocessor system based upon several MAP1000s.

5.4.2 VHDL models

A block diagram of our MAP1000 VHDL model is shown in Figure 5-5. The main components of the model are *processor core/cache*, *DMA channel*, an *SDRAM buffer* and *controller*, *PCI port*, an internal *IMB (I/O-memory bus) bus arbitrator*, and an external *PCI bus arbitrator*.

- The *processor core/cache* model runs an ATF file implementing five operations: read burst (*rb*), write burst (*wb*), no-address-operation (*an*), issue DMA transfer (*is*), and wait on DMA transfer (*wt*). With these five instructions, timing of the data flow with respect to the processor/cache and the control of the DMA channels can be simulated.

- The MAP1000's Data Streamer simulates six channels (instead of 64) since for running our ultrasound algorithms we utilize maximum of 6 channels. This helps to reduce the VHDL model complexity as well. Channels 0 and 1 are combined for input data flow from the SDRAM to the processor/cache; channels 2 and 3 are combined for output data flow from the processor/cache to either the SDRAM or PCI ports; and channels 4 and 5 are used for memory to memory transfers, such as when implementing the log LUT with guided transfers.

- The SDRAM models follow the specifications assumed by CASIM. The *SDRAM controller* simulates controlling of 2 bank, 2 kbytes per row SDRAM (64 bits at 100 MHz) and refreshes a row in each bank every 16 microseconds.

- The dual PCI ports (32 bits at 66 MHz) are modeled with each port having an 8 entry buffer for input and a 4 entry buffer for output. The *PCI Port* models act as agents to talk on the PCI bus, while *PCI channel* models act as agents to talk on the internal IMB bus. The *PCI arbitrator* uses round-robin arbitration, allowing a fixed 32-byte burst between bus masters.

- On the IMB internal bus, the *IMB arbitrator* uses a prioritized round-robin arbitration after 32-byte bursts, giving the highest priority to the *processor/cache* model, the next to the *PCI channel*, and the lowest priority to the *DMA channels*.

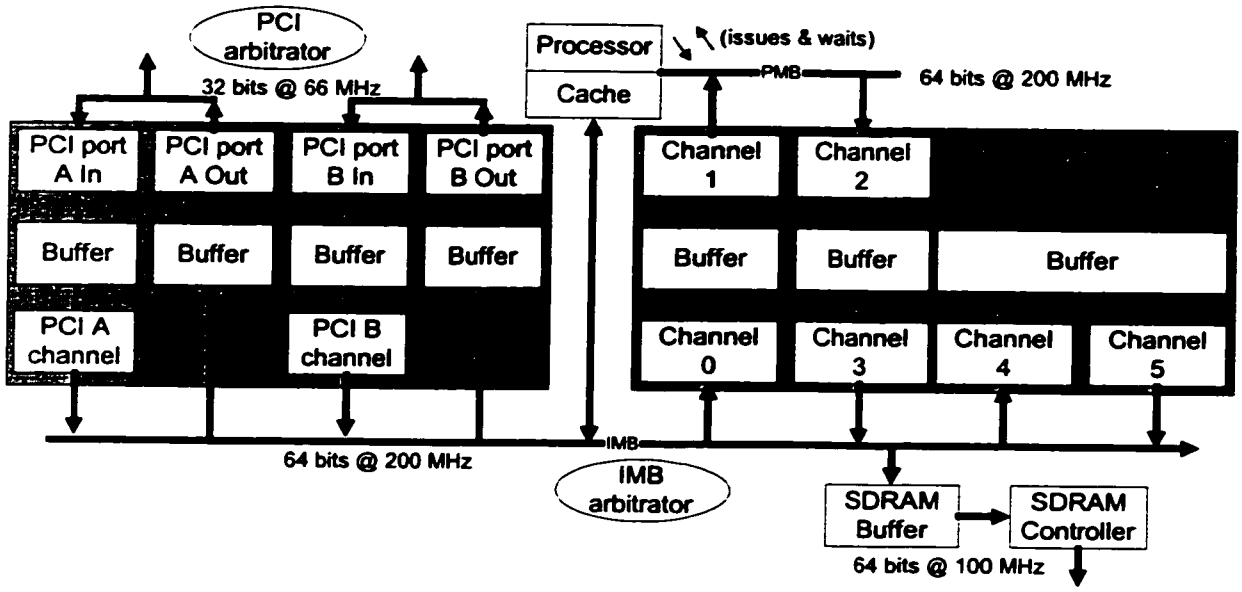


Figure 5-5. MAP1000 VHDL model.

For performing multiprocessor simulation several of MAP1000 modules are put together in the simulation environment with each processor connected to another processor through PCI bridge for interprocessor communication.

We rely on the accuracy of CASIM to simulate the timing of data accesses between the core processor and cache. We use a standard simulation environment (by Viewlogic) for simulating VHDL behavioral models and multiprocessor system, providing the accuracy and security of a well-established simulation tool. The multiprocessor performance is obtained by tracking key signals in the output timing diagrams. The computation load of each processor is known by the start and stop times of main computation loops and when the issues and waits on the DMA occur. In

addition, special VHDL models track the PCI bus statistics needed to determine bus loading. To validate the accuracy of the VHDL model and parser, we compared their accuracy to that of CASIM in executing various ultrasound algorithms with the dilation artifact in % shown in Table 5-3, indicating the VHDL simulator dilates the CASIM simulation by 0.55 to 2.46% for the ultrasound functions.

Table 5-3. Validation results, comparing the accuracy of the VHDL models to that of the CASIM simulator.

Program	% error
EP	1.24
SC	1.99
CF	0.55
EP1/CF	2.46
EP2/SC/FI/TF	1.85

The above validation is for the single MAP1000s, not the complete multiprocessor environment, which can only be verified after the multiprocessor hardware is prototyped. Since CASIM does not simulate the PCI ports, the above simulations do not verify the accuracy of the VHDL PCI ports and PCI arbitrator. The PCI model was verified by comparing the output waveforms of the PCI bus with those of the PCI standard specification [64].

5.5 ***Results and discussion***

The ultrasound algorithms mapped to the individual MAP1000 in Chapter 4 were mapped to the multiple processor architecture in both parallel and pipelined fashion, balancing the processing load equally across all the processors. Each board was assigned to process ½ of the input image in parallel fashion. Each individual board in turn processes the input (1/2) image in two stages in pipelined fashion. In the B mode (scenario #1), three processors perform echo processing in the first stage and one processor perform scan conversion in the second stage. In the color mode (scenario #6), two processors perform color flow processing and the magnitude with log compress part of echo processing in

first stage and the other two processors perform echo processing filters, scan conversion, and tissue/flow decision in the second stage. Dividing the image processing spatially incurs the overhead of recomputing overlapping lateral vectors needed between the image sub-sections, due to the neighborhood operators such as scan conversion, convolution, and median filter. Figure A-1 shows the simulated timing diagrams illustrating the pipelining of the EP on three processors and SC on the fourth processors for B-mode (for a single board) while Figure A-3 shows the simulated timing diagram illustrating CF and *EP part 1* on the first two processors and *EP part 2* and SC on the next two processors for color-mode. For B-mode timing diagrams, three frames are shown and for color-mode single frame is shown (due to large compute time). Although at times all four processors appear to have conflicting transfers on the middle PCI bus for B-mode in Figure A-1 (and Figure A-3 for color-mode), a "zoomed-in" plot in Figure A-2 shows that the actual PCI data transfers are relatively sparse, thus bus conflicts do not occur as often as Figure A-1 imply.

Table 5-4 shows the simulation results for the worst case B-mode scenario (512 x 1024 at 68 fps) and Table 5-5 shows the simulation results for the worst case color-mode scenario (i.e., B = 62.5 fps and color = 15.6 fps). From the Table 5-4, the maximum processing load during B-mode occurs for the processors performing EP and it is about 85.6% (processing load for single PCI and dual PCI almost remains the same). In the dual-PCI port architecture, the bus-loading for the middle-bus is 29.9% through which most of the inter-processor data transfer occur. Thus there is sufficient bandwidth

Table 5-4. B-mode multiprocessor simulation results.

B-Mode	Clock (MHz)	Bus width	Dual-PCI bus board	Single-PCI bus board
PROCESSING LOAD				
EP	200	-----	85.6%	85.7%
SC	200	-----	76.6%	77.1%
BUS LOAD				
System bus	33	32 bits	30.2%	30.2%
System bus	33	64 bits	17.2%	17.2%
Middle bus	66	32 bits	29.9%	64.1%
Bottom bus	66	32 bits	32.4%	-----

available (~1/3 utilized) for inter-processor communication and data transfer. However for the single-PCI port system, the middle bus is heavily utilized (64% loaded) and offers less room for growth, but can still support the system specifications. For color mode (Table 5-5), the PCI buses are even less loaded than the B mode, with sufficient bandwidth for both the dual-PCI port and single-PCI port architectures still available. However, the processors are heavily loaded and the two processors performing *EP part1* and *CF* are 93% loaded and the processors performing *EP part 2*, *SC*, *FI* and *TFD* are 91% loaded.

Table 5-5. Color-mode multiprocessor simulation results

Color Mode	clock (MHz)	Bus Width	Dual-PCI bus board	One PCI bus board
PROCESSING LOAD				
EP1/CF	200	-----	93.2%	93.7%
EP2/SC/FI/TF	200	-----	91.0%	91.1%
BUS LOAD				
System bus	33	32 bits	30.1%	30.1%
System bus	33	64 bits	17.3%	17.3%
Middle bus	66	32 bits	18.6%	46.1%
Bottom bus	66	32 bits	27.8%	-----

The results indicate that we could meet our system requirement but the major concern is the processing load which range from 85.6% to 93.2%. However these were for stringent worst case scenarios e.g., maximum frame rate of 68 fps for B mode, a large data size (input 512 x 512 and output 800 x 600), and dual-beams. In addition, in color mode, the ratio between B-mode frame rate and color flow frame rate was set to 4 even though clinicians normally use the ratio of 1. By scaling down the system requirement to match the specifications of the current ultrasound machines, we can decrease the processing load on our system significantly. For example, if we scale down the B-mode frame rate requirement to 40 fps for the large 512 x 1024 input data size, the maximum processing load reduces to 51%. Similarly by limiting the ratio between B frame rate and

color flow frame rate in color mode to one and maximum frame rate to 16 (with the ensemble size of 6) we can reduce the maximum processing load to 59%.

One more system specification that can be changed to reduce the processing load but still have no degradation in the final output image is the number of input samples per vector. In chapter 2, the number of samples per vector is fixed to 1024 for B-mode and 512 for color-mode independent of the depth of imaging since we are assuming that the sampling frequency is fixed and always generates fixed number of samples. However For 20 kHz PRF, this specification is overspecified. For example, the axial resolution for B-mode and color-mode mode data when f_0 is 7.5 MHz (in Eq. (2-8)) is 0.1 mm and 0.4 mm (N=2 for B–mode and N=4 for color mode) respectively. For the 20 kHz PRF, the vector depth is ~3 cm resulting in 300 samples per vector for B data and 75 samples per vector for color data (with 1 sample per range bin), which are much less than our specification of 1024 and 512, respectively. By utilizing programmable A/D converters we can control the number of output samples per vector depending upon the PRF, thus reducing the number of samples per vector significantly, which in turn reduces the processing load. If we utilize 2x sampling (x being the minimum number of samples) for this 7.5 MHz transducer, for B mode the processing load on the first stage reduces to 50% and on the second stage reduces to 70% and for color mode the processing load, on first stage reduces to 54.6% and on the second stage reduces to 77.9%. Thus, a reasonable system load lies somewhere between 51 % and 86% for B-mode and 59% to 94% for color mode, providing a safe margin for the system design.

The estimated cost for this 4-processor board is around $1400, or $2800 for a two-board system. This programmable system replaces 9 uniquely designed boards totaling over $10,000 in a commercial ultrasound machine. Thus, the programmable approach not only greatly reduces the system cost, but also can potentially reduce the non-recurring engineering cost by developing only on one board (repeated throughout the system) instead of 9 custom boards.

Chapter 6: Conclusions and Future Directions

6.1 Conclusions

Ultrasound imaging has become a popular imaging modality because it is safe, non-invasive, relatively inexpensive, easy to use, and capable of real-time imaging. In order to meet the high computation and throughput requirements, ultrasound machines have been traditionally designed using algorithm-specific fixed-function hardware with limited reprogrammability. As a result, improvements to the various ultrasound algorithms and additions of new ultrasound applications have been quite expensive, requiring redesigns ranging from ASIC chips and boards up to the complete machine. On the other hand, a fully programmable ultrasound machine could be reprogrammed to quickly adapt to new tasks and offer advantages, such as reducing costs and the time-to-market of new ideas. Additionally, the programmable system would provide a real-time platform to experiment many new ideas, features, and applications. Despite these advantages, an embedded programmable system capable of meeting the processing requirements of a modern ultrasound machine has not yet emerged due to insufficient compute power [6, 9, 66], inadequate data flow bandwidth or topology [5], or algorithms not optimized for the architecture. This study has addressed the issues associated with proving the feasibility of a fully programmable ultrasound system not only by developing the architecture capable of handling the computation and data flow requirements, but also designing tightly-integrated ultrasound algorithms. Finally, we demonstrated by using a unique simulation method that our system meets the requirements.

*6.2 **Contributions***

In this research the feasibility of a cost-effective, programmable ultrasound color flow system capable of handling the real-time processing requirements was demonstrated. The contributions include:

(1) ***Multiprocessor Architecture for Ultrasound***: We designed a 2-board system composed of 8 mediaprocessors. The simulation results showed that this 2 board system meets both the computation and bus load requirement of all the existing color flow system modes. In addition, since our system is programmable, the same hardware can be reused in developing new modes, e.g., 3D and panoramic imaging. The flexibility of the our programmable system allows us to reconfigure the system to support different algorithms, e.g., velocity estimation using cross correlation technique instead of the more widely-used autocorrelation technique whose requirements are very different. We have 64-Mbytes of SDRAM memory on each processor, which can be utilized to store CINE frames at various processing stages. This helps to modify various parameters, such as filtering coefficients, persistence, and scan conversion parameters (pan, zoom etc), on the fly. Since we use a common board repeated through the system with low-cost mediaprocessors, standard SDRAM memory chips and a standard bus, it can be easily scaled by adjusting the number of boards.

(2) ***Algorithm Mapping Techniques:*** Using powerful mediaprocessors does not guarantee high performance for the algorithms. Efficient algorithms that are tightly coupled to the mediaprocessor architecture are needed to implement the entire system with reasonable number of processors. We developed several methodologies that can be utilized to obtain high performance, e.g., (a) mapping algorithms to utilize powerful instruction sets, (b) remove barriers to data-level parallelism, such as *if/then/else*, (c) utilize loop unrolling and software pipelining, and (d) utilize a DMA controller to reduce the I/O overhead. We have

developed a methodology to determine the efficiency of individual algorithms and showed how $t_{compute}$ and $t_{i/o}$ estimates can be utilized for an optimal system.

(3) ***Ultrasound Algorithm Mapping Studies:*** We developed several ultrasound algorithms mapped efficiently to the underlying mediaprocessor architecture. Our 2D convolution algorithm mapped to the MAP1000 architecture has performance comparable to and in many cases faster than several hardwired approaches. We developed a new partitioned CORDIC algorithm to compute *arctangent* and *square root*, which has an advantage of using very little on-chip memory and has performance faster than cache-based method when the number of bits per pixel is less than 11 bits. We presented how computationally expensive wall filters can be implemented efficiently on the mediaprocessor. For an IIR filter, we developed a unique technique to overcome the limitations of interdependency between previous and current output. We efficiently utilized the overflow bit of 2's complement to implement a computationally-expensive circular median filter. We also showed how other algorithms, such as *autocorrelation*, *clutter_shift*, could be implemented utilizing powerful mediaprocessor instructions. For an optimal system, implementation of individual algorithms separately may not be sufficient since many algorithms are I/O-bound. We showed how several algorithms can be combined to share the I/O, thus improving the system performance significantly.

(4) ***Multiprocessor Simulation Method***: Since simulations are necessary to demonstrate that the multiprocessor system architecture meets the processing load and bus bandwidth requirements, we developed a unique multiprocessor simulation environment. We utilized the cycle accurate simulation (CASIM) of the MAP1000 (developed by Equator Technologies) to generate our address trace files. These address trace files in combination with VHDL models remove the complexity of processor modeling, thus reducing the simulation time while maintaining the accuracy. This work was done in collaboration with Dr. George

York who developed the MAP1000 model to run the generated address trace files.

6.3 Future directions

Having designed and simulated the programmable ultrasound architecture and demonstrated in detail its feasibility, the next step is to see this system commercialized to realize its benefits and contribute to improving the quality of health care. Areas of future work include mapping advanced ultrasound applications, reducing risks in developing the commercial high-end color flow system, and staying current with the mediaprocessor advances (or developing more powerful mediaprocessor that are easier to program than the current mediaprocessor).

6.3.1 Advanced ultrasound applications

Previously it was demonstrated that the PUIP system (composed of 2 TMS320C80s) could adapt and handle the real-time processing load of several new advanced features, such as panoramic imaging, segmentation/quantitative imaging, and 3D imaging [13]. The third board with 4 MAP1000s (in addition to 2 boards with 4 MAP1000 each) can provide much more computing power than the PUIP's 2 TMS320C80s (25.6 BOPS versus 5.6 BOPs). Thus, our system can in addition to all the applications the PUIP could handle offer room for improvement, e.g., better motion estimation for panoramic imaging and larger 3D volumes or faster reconstruction for 3D imaging. For applications that often process after image acquisition, such as 3D volume rendering or automatic segmentation of image features (e.g., such as pubic arch, fetal head, and epicardial and endocardial boundaries [8]), an additional board is not required as all 8 MAP1000s are free to be utilized for post-acquisition processing. For example, current 3D volume rendering systems use small volume, e.g., 128^3 = 2 Mbytes or 256^3 = 16 Mbytes, which can easily fit in the SDRAM available on one MAP1000. Volumes of 512^3 or 128 Mbytes will not only require the memory space and processing power available on our

multiple MAP1000 boards, but also new algorithms to share volume reconstruction, rendering computations and 3D data sharing across multiple processors. The MAP1000 being a video processor can perform MPEG-1 and MPEG-2 compression and decompression in real time [47]. This means that our system can perform teleradiology and telemedicine applications to generate in real time the compressed digital video bitstreams via H.261 or H.263 for the videoconferencing video bitstreams while better quality compression algorithms, such as MPEG-2 for the diagnostic bit streams. In addition, since it is fully programmable, proprietary algorithms can also be implemented to have a better compression ratio, such as compressing the pre-scan converted data using wavelets [67]. Thus our programmable system will help to bring forth several enabling technologies for improved diagnosis, lower cost, and more widespread use.

6.3.2 Developing the commercial programmable system

There are several architectural issues involved in developing this programmable system. One important issue is the total power requirement of each board. Since the PCI specifications limit each PCI board to 25 W of power consumption, our architecture with four processors each with 64-MByte SDRAM, plus 2 PCI bridges on the board, will require more than 25 W (currently each MAP1000 is rate to consume 6 W). Thus, an additional power source (and cooling) might be required. The architectural issue will be developing the board layout with four MAP1000s plus two arbitrators connected across on-board PCI bus running at 66 MHz. Systems designed with these high-speed buses must be carefully designed, ensuring the maximum length between nodes is short enough for the signals to propagate within this short clock period. Failure to consider these factors resulted in several early systems targeted for 66 MHz PCI bus only achieving 40-50 MHz [68]. The third issue is to design a real-time multi-tasking operating system to automatically reconfigure and balance processor load for the system when the clinician changes the mode of operation, transducers, or other settings.

Thus, there are several issues and risks involved in developing the programmable system using 8 MAP1000s. The approach that can be utilized to reduce the risks in

designing the high-end programmable system is to design a low-end ultrasound machine by reducing system specifications and later scale it up for a high-end system. A sister processor of MAP10000 called MAP-CA can be utilized. The MAP-CA consumes less power, runs at 300 MHz, supports up to 128 Mbytes of SDRAM, has larger data and instruction cache (32 kbytes each) and is less expensive. The increased processor frequency and cache size would reduce the number of processors required to implement the system. However, the disadvantages are that it does not have a floating-point unit and has a single PCI port. Since we can avoid floating-point operations using fixed-point operations in most of our implementations, porting to this processor will not be a problem. However, the major limitation comes from the availability of a single PCI port rather than two. Currently, we are pursuing this approach by developing a low-end machine in using the MAP-CA processors.

6.3.3 Selection of a processor for a programmable system.

Current mediaprocessors will be improved and upgraded by new mediaprocessors offering better architectures and/or higher clock speeds e.g., Intel's Merced [69]. Since the future mediaprocessors are likely to continue the trend toward supporting the instruction level parallelism and data-level parallelism, and efficient I/O handling mechanism, our algorithms and algorithm mapping techniques and the algorithms can be readily utilized on newer processors as well. However, before selecting a processor for a target system, both the mediaprocessor study similar to what we performed for our system design and the algorithm study should be performed. Even though more computation power is will be available by increasing the clock frequency, the I/O will remain as a key bottleneck. The processor memory bandwidth usually does not scale with the increase in processor's clock speed. As many ultrasound stages are already I/O-bound, such as autocorrelation, the increase in computation power may not be of significant help in reducing the total number of processors required in designing the system. Thus, a system's approach is needed to select a mediaprocessor, estimate the

number of processors, and develop an ultrasound architecture composed of new mediaprocessors.

Bibliography

1. C. Basoglu, R. Managuli, G. York, and Y. Kim, "Computing requirements of modern medical diagnostic ultrasound machines," *Parallel Computing*, vol. 24, pp. 1407-1431, 1998.
2. T. Loupas, W. N. McDicken, T. Anderson, and P. L. Allan, "Development of an advanced digital image processor for real-time speckle suppression in routine ultrasonic scanning," *Ultrasound Med. Biol.*, vol. 20, pp. 239-249, 1994.
3. G. Pasterkamp, C. Borst, A. S. R. Moulaert, C. J. Bouma, D. VanDijk, M. Kluytmans, and B. M. ter-Haar-Romeny, "Intravascular ultrasound image subtraction: a constrast enhancing technique to facilitate automatic three-dimensional visualization of the arterial lumen," *Ultrasound Med Biol.*, vol. 21, pp. 913-918, 1995.
4. K. Rosenfield, J. Kaufman, M. Peiczek, R. E. Langevin, E. Palefski, S. A. Razvi, and J. M. Isner, "Human coronary and peripheral arteries: On-line three-dimensional reconstruction from two-dimensional intravascular US scans," *Radiology*, vol. 184, pp. 823-832, 1992.
5. P. Jensch and W. Ameling, "Analysis of ultrasound image sequences by a data-flow architecture supporting concurrent processing," *SPIE Hybrid Image and Signal Processing*, vol. 939, pp. 229-236, 1988.
6. J. L. Jensen, J. A. Jensen, P. F. Stetson, and P. Antonius, "Multiprocessor system for real-time deconvolution and flow estimation in medical ultrasound," *IEEE Ultrason. Symposium Proc.*, vol. 2, pp. 1197-200, 1996.
7. L. N. Bohs, B. H. Friemel, B. A. McDermott, and G. E. Trahey, "A real time system for quantifying and displaying two-dimensional velocities using ultrasound," *Ultrasound Med. Biol.*, vol. 19, pp. 751-761, 1993.

8. Y. Kim, J. H. Kim, C. Basoglu, and T. C. Winter, "Programmable ultrasound imaging using multimedia technologies: A next-generation ultrasound machine," *IEEE Trans. Information Tech. Biomed.*, vol. 1, pp. 19-29, 1997.
9. Advanced Technology Laboratories, *ATL Announces New Breakthrough Product*, http://www.atl.com/news/55_pr_022097.html, 1997.
10. Medison, *World's First B/W Only Digital Ultrasound*, http://www.medison.co.kr/, 1999.
11. L. Weng L, A. P. Tirumalia, C. M. Lowery, L. F. Nock, D. E. Gustafson, P. L. Von-Behren, and J. H. Kim, "Ultrasound extended-field-of-view imaging technology," *Radiology*, vol. 203, pp. 877-880, 1997.
12. S. D. Pathak, V. Chalana, and Y. Kim, "Interactive automatic fetal head measurements from ultrasound images using multimedia computer technology," *Ultrasound Med. Biol.*, vol. 23, pp. 665-673, 1996.
13. W. S. Edwards, C. Deforge, and Y. Kim, "Interactive three-dimensional ultrasound using a programmable multimedia processor," *International J. Imaging Systems & Technology*, vol. 9, pp. 442-454, 1998.
14. J. A. Jensen, *Estimation of blood velocities using ultrasound*, Cambridge: Cambridge University Press, 1996.
15. S. Berg and H. Torp, "Volumetric blood flow estimation using dynamic three-dimensional ultrasound color flow imaging," *IEEE Ultrasonics Symposium Proceedings*, vol. 1892, pp. 1513 –1516, 1998.
16. F. Ratliff, *Mach Bands: Quantitative Studies on Neural Networks in the Retina*, San Fransisco, CA, Holden Day, pp. 139-140, 1965.
17. E. P. Novakov, "Online median filter for ultrasound signal processing," *Med. & Biol. Eng. & Comp*, vol . 29, pp.222-224, 1991.
18. A. N. Evans, M. S. Nixon, "Temporal methods for ultrasound speckle reduction," *IEE Texture Analysis in Radar and Sonar*, vol. 1, pp. 1-6, 1993.

19. D. S. Kalivas, A. A. Sawchuck, "Motion compensated enhancement of noisy image sequences," *Int. Conf. Acoustics, Speech, and Signal Processing,* Albuquerque, vol. 4, pp. 2121-2124, 1990.

20. W. Steinke, T. Els, and M. Hennerici, "Comparison of flow disturbances in small cartoid atheroma using a multi-gate pulsed Doppler system and Doppler color flow imaging," *Ultrasound Med. Bio.*, vol. 18, pp. 11-18, 1992.

21. C. Kimme-Smith, F. N. Tessler, E. G. Grant, and R. R. Perella, "Processing algorithms for color flow Doppler," *IEEE Ultrasonics*, vol. 2, pp. 877-879, 1989.

22. C. Kasai, K. Namekawa, A. Koyano, and R. Omoto, "Real-time two-dimensional blood flow imaging using an autocorrelation technique," *IEEE Trans. Sonics and Ultrasonics*, vol. 32, pp. 458-464, 1985.

23. W. D. Barber, J. W. Eberhard, and S. G. Karr, "A new time domain technique for velocity measurements using Doppler ultrasound," *IEEE Trans. Biomed. Engineering,* vol. 32, pp. 213-229, 1985.

24. D. A. Christensen, *Ultrasonic Bioinstrumentation,* New York: Wiley, 1996.

25. O. Bonnefous and P. Pesque, "Time domain-formulation of pulse-Doppler ultrasound and blood velocity estimation by cross correlation," *Ultrasonic Imaging* vol. 8, pp. 73-85, 1986.

26. K. W. Ferrara and G. DeAngelis, "Color flow mapping," *Ultrasound Med. Biol.*, vol. 23, pp. 321-345, 1997.

27. H. F. Routh, "Doppler ultrasound," *IEEE Engineering Med & Bio* vol. 15, no. 6, pp. 31-40, 1996.

28. A. P. Kadi and T. Loupas, "On the performance of regression and step initialized IIR clutter filters for color Doppler systems in diagnostical medical ultrasound," *IEEE Trans. Ultrason. Ferroelect. Freq. Control*, vol. 42, pp. 927-937, 1995.

29. A. P. G. Hoeks, J. J. W. van de Vorst, and A. Dabekaussen, "An efficient algorithm to remove low frequency Doppler signals in digital Doppler systems," *Ultrasonic Imaging*, vol. 13, pp. 135-144, 1991.

30. L. Thomas and A. Hall, "An improved wall filter for flow imaging of low velocity flow," *IEEE Ultrason. Symposium Proc., Cannes*, vol. 3, pp. 1701-1704, 1994.

31. J. Ophir and N. F. Makland, "Digital scan converters in diagnostic ultrasound imaging," *Proceedings of the IEEE*, vol. 67, pp. 654-664, 1979.

32. J. A. Parker and D. E. Troxel, "Comparison of interpolating methods for image resampling," *IEEE Trans. Med. Imag*, vol. 2, pp. 31-39, 1983.

33. D. A. Patterson and J. L. Hennessey, *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, San Francisco, CA, 1996.

34. J. A. Fisher, "The VLIW machine: A multiprocessor from compiling scientific code," *Computer*, vol. 17, pp. 45-53, July 1984.

35. R. Lee, "Accelerating multimedia with enhanced microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22-32, 1995.

36. C. Basoglu, D. Kim, R. J. Gove, and Y. Kim, "High-performance image computing with modern microprocessors," *International Journal of Imaging Systems and Technology*, vol. 9, pp. 407-415, 1998.

37. S. Rathnam and G. Slavenburg, "Processing the new world of interactive media. The Trimedia VLIW CPU architecture," *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 108-117, 1998.

38. K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, vol. 14, no. 4, pp. 59-67, 1994.

39. Berkeley Design Technology (BDT), "DSP processor fundamentals," http://www.bdti.com/products/dsppf.htm, 1996.

40. D. Kim, R. A. Managuli, and Y. Kim, "Data cache vs. direct memory access (DMA) in programming mediaprocessors," submitted to *IEEE Micro*, 2000.

41. P. Faraboschi, G. Dcsoli, and J. A. Fisher, "The latest world in digital media processing," *IEEE Signal Processing Magazine*, vol. 15, pp. 59-85, Mar. 1998.

42. N. Seshan, "High VelociTI processing," *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 86-101, 1998.

43. J. J. Shieh, and C. A. Papachristou, "Fine grain mapping strategy for multiprocessor systems," *IEE Proceedings in Computer Digital Technology*, vol. 138, pp. 109-120, 1991.

44. Texas Instruments, "TMS320C6211, fixed-point digital signal processor," http://www.ti.com/sc/docs/products/dsp/tms320c6211.html, 1999.

45. Fujitsu Limited, " FR500, " http://www.fujitsu.co.jp, 1999.

46. K. Guttag, R. J. Gove, and J. R. VanAken, "A single chip multiprocessor for multimedia: The MVP," *IEEE Computer Graphics and Application*, vol. 12, no. 6, pp. 53-64, 1992.

47. C. Basoglu, R. J. Gove, K. Kojima, and J. O'Donnell, "A single-chip processor for media applications: The MAP1000," *International Journal of Imaging Systems and Technology*, vol. 10, pp. 96-106, 1999.

48. M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN: Conference on Programming Language Design and Implementation*, vol. 23, pp. 318-328, 1988.

49. A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42-50, 1996.

50. G. York, R. Managuli, and Y. Kim, "Fast binary and gray-scale mathematical morphology on VLIW mediaprocessor," *SPIE Electronic Imaging*, vol. 3645, pp. 45-55, 1999.

51. LSI, *L64240 Multibit Filter (MFIR),* LSI Logic Corporation, Milpitas CA, 1989.

52. R. Managuli, G. York, D. Kim, and Y. Kim, "Mapping of 2D convolution on VLIW mediaprocessors for real-time performance," *Journal of Electronic Imaging,* vol. 9, pp. 327-325, 2000.

53. J. Kim and Y. Kim, "Efficient 2-D convolution algorithm with the single-data multiple kernel approach," *Graphical Models and Image Processing*, vol. 57, pp. 175-182, 1995.

54. R. A. Managuli, C. Basoglu, S. D. Pathak, and Y. Kim, "Fast convolution on a programmable mediaprocessor and application in unsharp masking," *SPIE Medical Imaging*, vol. 3335, pp. 675-683, 1998.

55. N. Nikolaidis and N. I. Pitas, "Nonlinear processing and analysis of angular signals," *IEEE Trans. Signal Processing*, vol. 46, pp. 3181-3194, 1998.

56. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronics Computers*, vol. 8, pp. 330-334, 1959.

57. D. Timmermann, H. Hahn, and B. Hosticka, "Low latency time CORDIC algorithms," *IEEE Transactions on Computers*, vol. 41, pp. 1010-1015, 1992.

58. H. Dawid and H. Meyr, "High speed bit-level pipelined architectures for redundant CORDIC implementation," *Proceedings of Int'l Conf. Application Specific Array Processors*, pp. 358-372, August 1992.

59. E. Antelo, J. Bruguera, J. Villalba, and E. Zapata, "Redundant CORDIC rotator based on parallel prediction," *Proceedings of Int'l Symp Computer Arithmetic*, pp. 172-179, July 1995.

60. S. Wang, V. Piuri, and E.E. Swartzlander Jr., "Hybrid CORDIC algorithms," *IEEE Transactions on Computers*, vol. 48, pp. 1202-1207, 1997.

61. G. York, *Architecture and Algorithms for a Fully Programmable Ultrasound System*, Ph.D. Dissertation, University of Washington, Seattle, 1999.

62. R. B. Peterson, L. E. Atlas, and K. W. Beach, "A comparison of IIR initialization techniques for improved color Doppler wall filter performance," *Ultrasonic Symposium proceedings*, vol. 3, pp. 1705-8, Nov. 1994.

63. C. B. Stunkel, B. Janssens, W. K. Fuchs, "Address Tracing for Parallel Machines," *IEEE Computer*, vol. 24, no. 1, pp. 31-38, 1991.

64. PCI Special Interest Group, *PCI Local Bus Specification, Revision 2.0*, Hillsboro, Oregon, 1993.

65. J. H. Kim, *Towards More Efficient Domain-Specific Image Computing*, Ph.D. Dissertation, University of Washington, Seattle, 1995.

66. C. Basoglu, *A generalized programmable system and efficient algorithms for ultrasound backend processing*, Ph.D. Dissertation, University of Washington, Seattle 1997.

67. J. E. Cabral, D. T. Linker, and Y. Kim, "Compression for pre-scan-converted ultrasound sequences," *Proceedings of the SPIE*, vol. 3335, pp. 378-387, 1998.

68. H. M. Needham, "Peripheral component interconnect (PCI) bus for ASIC designers," *Texas Instruments Application's Notes SRGA013*, http://www.ti.com/sc/docs/asic /srga013/s1.htm, 1995.

69. T. Cantrell, "Test driving a Merced with pins," *Circuit-Cellar-Ink*, no.113, p.78-82, December 1999.

APPENDIX A

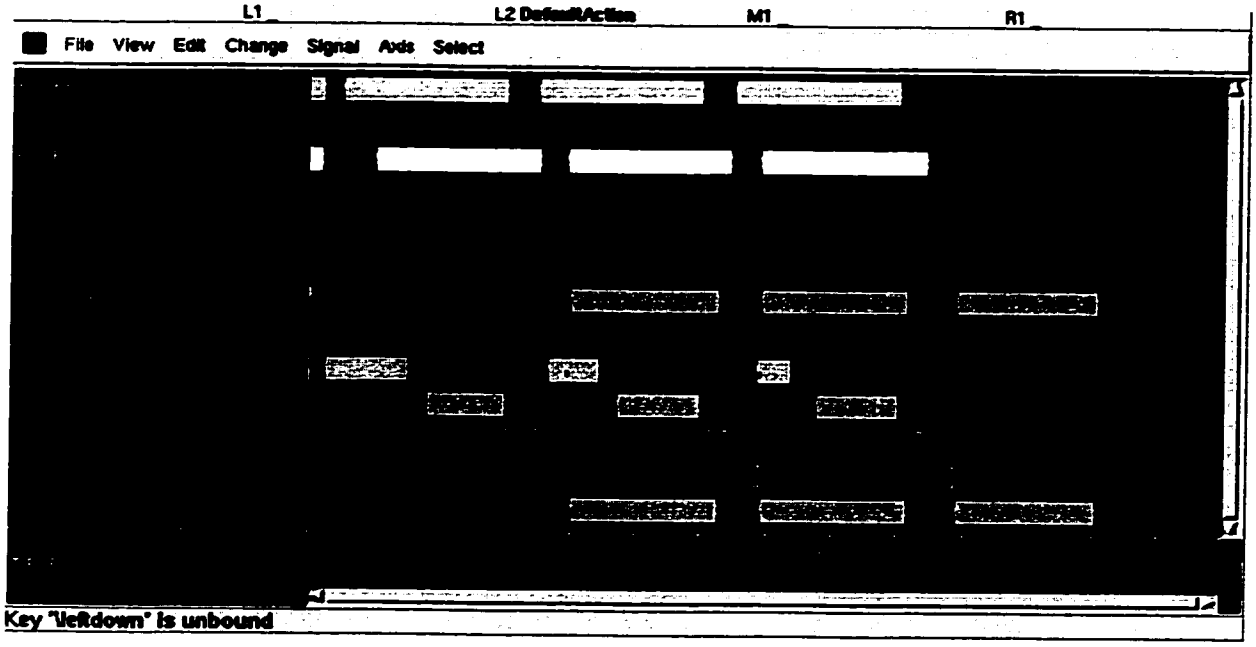


Figure A- 1. Processing load and PCI bus load for the middle bus, for 3 frames of B-mode simulation of the single PCI-port architecture. Signals P_OPER*x* show processing on the core processor (indicated by color bars) versus the wait periods (indicated by "08"), as does P_FRAME*x* with odd numbers indicating the processing time and even numbers indicating the wait periods. On signals P_PCI_OP*x*, solid bars and "05" indicate the time period a processor or bridge is trying to transfer a frame sub-section on the PCI bus. $x = 0$ for the bridge (sending vectors); $x = 1$, 2, or 3 for the EP processors; and $x = 4$ for the SC processor.

Figure A- 2. Zoomed-in PCI bus load for the middle bus, illustrating the bus conflicts during steady state of the B-mode simulation when all four processors are using the bus. "05" indicates a processor is trying to transfer a large block of data (e.g., ~1024 kbytes). Fortunately, each processor needs to use the bus relatively sparsely, having a plenty of time to finish one transfer before the next one begins.

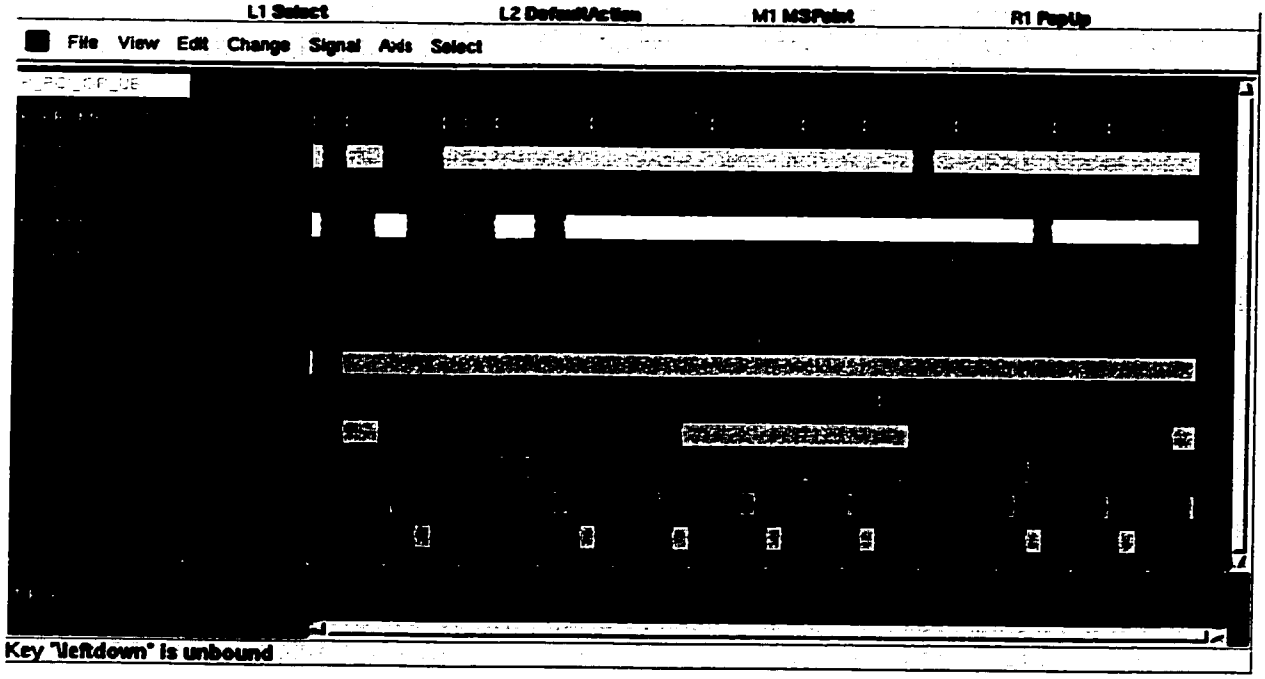


Figure A- 3. Color-mode simulation signals showing steady-state processing load (P_OPER*x* and P_FRAME*x*) and bus load (P_PCI_OP*x*), with K=4 (or 4 B frames for every color frame). *x*= 1 or 2 for EP1/CF processors; *x*= 3 or 4 for the EP2/SC/FI/TF processors; and *x*= UB for the bridge on the bottom PCI bus.

APPENDIX B: Example CASIM Debug File

Example entries from a CASIM debug file

The details of the events tracked cause excessively large files. For example, the total debug file size when simulating convolution of a 512x512 image with a 3x3 kernel is 420 Mbytes. Below are selected entries from an example CASIM debug file, showing key processor, data streamer, and SDRAM events, such as when the processor *kicked* a DS data transfer (cycle 12931), when the DS receives the transfer parameters (cycle 12947), the SDRAM row miss for the first data (cycle 12991), the RAS and CAS signals (cycles 12994 and 13001), and the data transfers for the first 32 bytes (cycles 13001-13009). Our *parser* tool creates the *ATF* files for our multiple processor simulations by extracting the key information from the CASIM debug file, greatly reducing the file size in the process.

```
cycle:12931: DS: DSHandle_Kick: #13348 channel 0 kicked off with
             Descriptor address 0x007fee60
...
cycle:12947: DS: DSReceiveDesc: descriptor got for channel 0,
             nextDescAddr 0x7fee60, dataAddr  0xb500, count 0x8,
             controlWord 0x38, pitch 0x0, width 0x200  PA:0x7fee60
...
cycle:12991: MB: SDRAMStateMachine: #13414 moving entry from
             pStage3Mess to pPrecharge ROW_MISS, setting
             SDRAMBusUsed=TRUE, no other request in SDRAM BA: 0xb500
...
cycle:12994: MB: SDRAMStateMachine: #6 moving entry from pP2 to pRAS,
             setting SDRAMBusUsed=TRUE BA: 0x100
...
cycle:13001: MB: firstCASCycle: #13414 moving LOAD to CAS, cyclesToGo
             3, cyclesToEarlyWarning 5  BA: 0xb500
cycle:13003: MB: dataToFromMemory: #13414 LOADING
             Data[0]=0xffffffffffffffff from CasimMemory[0x0000b500]
cycle:13005: MB: dataToFromMemory: #13414 LOADING
             Data[1]=0xffffffffffffffff from CasimMemory[0x0000b508]
cycle:13007: MB: dataToFromMemory: #13414 LOADING
             Data[2]=0xffffffffffffffff from CasimMemory[0x0000b510]
cycle:13009: MB: dataToFromMemory: #13414 LOADING
             Data[3]=0xffffffffffffffff from CasimMemory[0x0000b518]
```

VITA

Ravi Managuli

Academic Degrees

Ph.D. in Electrical Engineering, University of Washington, Seattle, 2000

Dissertation: *Programmable ultrasound color flow system.*

MS in Electrical Engineering, University of Nevada, Las Vegas, 1996

Thesis: *Capacitor start induction motor.*

BS in Electrical Engineering, Regional Engineering College, Warangal, India, 1993

Books

Ravi Managuli and Yongmin Kim, "VLIW processor architecture and algorithm mapping forDSP applications," a chapter in a book *Programmable Digital Signal Processors: Architecture, Programming, and Applications,* in press, 2000.

Journal Papers

Ravi Managuli, George York, and Yongmin Kim, "Fully programmable ultrasound machine using multimedia technologies," to be submitted to *IEEE Trans. on Information Technology in Biomedicine*, December 2000.

Donglok Kim, Ravi Managuli, and Yongmin Kim, "Use of on-chip memory vs. data cache in programming mediaprocessors," submitted to *IEEE Multimedia,* February 2000.

Ravi Managuli, George York, Donglok Kim, and Yongmin Kim, "Mapping of 2D convolution on VLIW mediaprocessors for real-time performance," *Journal of Electronic Imaging*, vol. 9, pp. 327-335, 2000.

C. Basoglu, R. Managuli, G. York, and Y. Kim, "Computing requirements of modern medical diagnostic ultrasound machines," *Parallel computing*, vol. 24, pp 1407-1431,

1998.

Conference Papers

G. York, R. Managuli, and Y. Kim, "Fast Binary and Gray-scale Mathematical Morphology on VLIW Mediaprocessors," *SPIE Electronic Imaging*, vol. 3645, Jan 1999, pp 45-55.

R. Managuli, G. York, and Y. Kim, "An Efficient Convolution Algorithm for VLIW Mediaprocessors," *SPIE Electronic Imaging*, vol. 3655, Jan 1999.

R. Managuli, C. Basoglu, S. D. Pathak, and Y. Kim, "Fast convolution on programmable microprocessor and application in unsharp masking," *Proc. of SPIE conf. medical imaging*, vol. 3335, pp. 675-685, 1998.

Technical Reports

"Architecture and algorithms for a fully programmable ultrasound machine," *Siemens Medical Systems Ultrasound Group*, Seattle, WA, August 1999.

"Mapping of ultrasound algorithms onto a mediaprocessor-based architecture: First quarterly report," *Hitachi Medical Corporation*, Japan, April 2000.

"Mapping of ultrasound algorithms onto a mediaprocessor-based architecture: Second quarterly report," *Hitachi Medical Corporation*, Japan, August 2000.